06-12-00

A

00/60/90 JC862 U.S. PTO

| NEW UTILITY PATENT APPLICATION TRANSMITTAL | DOCKET NO. 22727/04060 |
|---|---|
| | TOTAL PAGES IN THIS SUBMISSION 123 |

JC829 U.S. PTO 09/591432 06/09/00

**TO THE ASSISTANT COMMISSIONER FOR PATENTS**
**Box Patent Application**
**Washington, D.C. 20231**

Transmitted herewith for filing under 35 USC 11(a) and 37 CFR 1.53(b) is a new utility patent application for an invention entitled:

**SYSTEM AND METHOD FOR DETERMINING A SEED VIGOR INDEX FROM GERMINATED SEEDLINGS BY AUTOMATIC SEPARATION OF OVERLAPPED SEEDLINGS**

and invented by:

Miller Baird McDonald, Jr., Kikuo Fujimora , Mark Alan Bennett, Yusaku Sako, and Andrew Frederick Evans

**If a CONTINUATION APPLICATION, *check appropriate box and supply the requisite information:***
☐ **Continuation** ☐ **Divisional** ☐ **Continuation -in -part (CIP)**
**in prior application No.:** _____

Enclosed are:

Application Elements

1. ☒ Filing fee as calculated and transmitted as described below

2. ☒ Specification having 52 pages and including the following:
   - a. ☒ Descriptive Title of the Invention
   - b. ☐ Cross References to Related Applications *(if applicable)*
   - c. ☐ Statement Regarding Federally-sponsored Research/Development *(if applicable)*
   - d. ☐ Reference to Microfiche Appendix *(if applicable)*
   - e. ☒ Background of the Invention
   - f. ☒ Brief Summary of the Invention
   - g. ☒ Brief Description of the Drawings *(if drawings filed)*
   - h. ☒ Detailed Description
   - i. ☒ Claim(s) as Classified Below
   - j. ☒ Abstract of the Disclosure

3. ☒ Drawing(s) *(when necessary as prescribed by 35 USC 113)*
   - a. ☒ Formal
   - b. ☐ Informal

   Number of Sheets 10 .

4. ☒ Oath or Declaration
   - a. ☐ Newly executed *(original or copy)* ☒ Unexecuted
   - b. ☐ Copy from a prior application (37 CFR 1.63(d)) *(for continuation/divisional application only)*
   - c. ☒ With Power of Attorney ☐ Without Power of Attorney

| NEW UTILITY PATENT APPLICATION TRANSMITTAL | DOCKET NO. 227270/04060 |
|---|---|
| | TOTAL PAGES IN THIS SUBMISSION 123 |

Application Elements (Continued)

5. ☐ Incorporation by Reference (*usable if Box 4b is checked)*
The entire disclosure of the prior application, from which a copy of the oath or declaration is supplied under Box 4b, is considered as being part of the disclosure of the accompanying application and is hereby incorporated by reference therein.

6. ☐ Computer Program in Microfiche (*Appendix)*

7. ☐ Nucleotide and/or Amino Acid Sequence Submission (*if applicable, all must be included)*
   a. ☐ Paper Copy
   b. ☐ Computer Readable Copy (*identical to computer copy)*
   c. ☐ Statement Verifying Identical Paper and Computer Readable Copy

## Accompanying Application Parts

8. ☐ Assignment Papers (*cover sheet & document(s))*

9. ☐ 37 CFR 3.73(B) Statement *(when there is an assignee)*

10. ☐ English Translation Document *(if applicable)*

11. ☐ Information Disclosure Statement/PTO-1449 ☐ Copies of IDS Citations

12. ☐ Preliminary Amendment

13. ☒ Return Receipt Postcard

14. ☒ Certificate of Mailing
   ☐ First Class ☒ Express Mail (*Specify Label No.):* *EL084652689US*

15. ☒ Small Entity Statement(s)
   ☐ Statement filed in prior application; Status still proper and desired.

16. ☒ Additional Enclosures *(please identify below):*

Listing 1, Listing 2 and Listing 3 (source code for a majority of the seedling analysis)

| NEW UTILITY PATENT APPLICATION TRANSMITTAL | DOCKET NO. 22727/04060 |
|---|---|
| | TOTAL PAGES IN THIS SUBMISSION 123 |

FEE CALCULATION AND TRANSMITTAL

## CLAIMS AS FILED

| 1) For | 2) Number Filed | 3) Number Extra | Rate | Additional Fee |
|---|---|---|---|---|
| TOTAL CLAIMS (37 CFR 1.16(c)) | = | | x $ | $ |
| INDEPENDENT CLAIMS (37 CFR 1.16(c)) | = | | x $ | $ |
| First Pres. of Multiple Dep. Claims | | = 0 | | |
| Terminal Disclaimer | | = 0 | | |
| | | | Basic Filing Fee | $ |
| | | | TOTAL | $ |

☐ A check in the amount of _____$_____ to cover the filing fee is enclosed.
☐ The Commissioner is hereby authorized to charge and credit Deposit Account No. _____ as described below. A duplicate copy of this sheet is enclosed.
☐ Charge the amount of _____ as filing fee.
☐ Credit any overpayment or any deficiency.

Respectfully submitted,

Date: 6/9/2000

By: Sean T. Moorhead

Sean T. Moorhead, Reg. No. 38,564
Calfee, Halter & Griswold LLP
1400 McDonald Investment Center
800 Superior Avenue
Cleveland, Ohio 44114
(216)622 - 8844

EXPRESS MAILING CERTIFICATE
"EXPRESS MAIL" Mailing Label No.: EL084652689US
Date of Deposit : June 9, 2000
I hereby certify that this paper or fee is being deposited with the United States Postal Service "Express Mail Post Office to Addressee" service under 37 CFR 1.10 on the date indicated above and is addressed to the Assistant Commissioner for Patents, Box Patent Application, Washington, D.C. 20231
Typed or printed name of person signing this certificate:
Kurt Feuerstein

Signed: _____

'Express Mail" Mailing Label No. EL084652895US

Date of Deposit June 9, 2000

I hereby certify that this paper or fee is being deposited with the United States Postal Service "Express Mail Post Office to Addressee" service under 37 C.F.R. § 1.10 on the date indicated above and is addressed to the Commissioner of Patents and Trademarks, Washington, DC 20231.

Kurt Feuerstein
(Typed or printed name of Sender)

(Signature)

# SYSTEM AND METHOD FOR DETERMINING A SEED VIGOR INDEX FROM GERMINATED SEEDLINGS BY AUTOMATIC SEPARATION OF OVERLAPPED SEEDLINGS

## Copyright Notice



## Field of the Invention

The present invention relates generally to the field of seeds and systems and methods for determining seed vigor, and more specifically to a system and method for automatically determining a seed vigor index by analysis of a scanned image of a plurality of seedlings grown from a lot of seeds, including the ability to automatically separate and analyze overlapped seedlings.

## Background of the Invention

Seed vigor testing is a procedure for evaluating the quality of a seed lot based on visually quantifiable cues, such as speed and uniformity of seedling growth.

Various specifications for seed vigor testing exist, including the Association of Official Seed Analysts Vigor Testing Handbook (1983) and International Seed Testing Association Seed Vigour Testing Handbook (1987).

Although such tests provide meaningful results to the seed community, they are not routinely performed because of labor intensiveness and subjectivity, which typically varies from seed analyst to seed analyst, even among Registered Seed Technologists (RSTs). Such subjectivity also has hindered the performance of meaningful research into seed vigor, preventing widespread use of seed vigor testing beyond a few key crops. Additionally, the known existing systems for automatically determining seed vigor are overly simple and/or prohibitively expensive, requiring costly cameras and/or special hardware such as special germination chambers.

## Summary of the Invention

The present invention provides a system and method for automatically determining seed vigor index by analysis of a scanned image of a plurality of seedlings grown from a lot of seeds.

According to one aspect of the current invention, seedling analysis software is used to analyze an image of seedlings. The seedling analysis software preferably analyzes both hypocotyl and radicle lengths and thus determines the separation point between the two for each seedling. The seedling analysis software also preferably separates overlapped seedlings, preferably using a simulated annealing technique.

According to another aspect of the present invention, a low-cost scanner placed in an inverted configuration in a scanner enclosure is used to generate high-quality, reproducible images of seedlings.

According to yet another aspect of the present invention, a method of using ordinary germination boxes, i.e., "sandwich boxes" is used to germinate seedlings that

greatly facilitate computer-based analysis. In general, this method comprises placing germination blotter paper in the lid of a sandwich box and growing seedlings within the sandwich box in a nearly vertical (upright) position in a darkened germination chamber. The resulting seedlings produce hypocotyls that grow essentially upward and radicles that grow essentially downward, which greatly facilitates image acquisition of the entire seedling, e.g., with an inverted scanner. Additionally, according to this aspect of the present invention, seeds are gently pressed into the moistened blotter paper(s) to help them remain in place when the sandwich box is placed in the nearly vertical position. Additionally, because the seedlings are grown in the lid of the sandwich box, the lid can be separated from the rest of the sandwich box to facilitate image acquisition. In the case of an inverted scanner, the lid and seedlings are placed in a lid-holder, which slides the lid and seedlings under the inverted scanner to facilitate scanning.

It is therefore an advantage of the present invention to provide a system and method for determining a seed vigor index using seedlings grown in the lid of a sandwich box to facilitate image acquisition.

It is also an advantage of the present invention to provide a system and method for determining a seed vigor index using seedling analysis software that analyzes both hypocotyl and radicle lengths as determinants of seed vigor.

It is therefore another advantage of the present invention to provide a system and method for determining a seed vigor index using seedling analysis software that separates overlapped seedlings, preferably with simulated annealing.

It is a further advantage of this invention to provide a system and method for determining a seed vigor index using seedlings grown such that their hypocotyls and radicles are substantially parallel to each other to facilitate image acquisition and analysis.

It is yet another advantage of the present invention to provide a system and method for automatically determining a seed vigor index.

It is yet another advantage of the present invention to provide a system and method for automatically determining a seed vigor index using seedling analysis software

that determines a separation point for hypocotyl and radicle for a given seedling, e.g., by analyzing root hairs and secondary roots.

It is still another advantage of the present invention to provide a system and method for automatically determining a seed vigor index using seedling analysis software that analyzes the ratio of the radicle length to the hypocotyl length.

It is yet another advantage of the present invention to provide a system and method for automatically determining a seed vigor index using seedling analysis software that analyzes uniformity of seedlings, e.g., uniformity of hypocotyl length, uniformity of radicle length, uniformity of total length, uniformity of the ratio of the radicle length to the hypocotyl length, and/or the number of dead seeds, and preferably that analyzes standard deviation of hypocotyl length, standard deviation of radicle length, standard deviation of total length, standard deviation of the ratio of the radicle length to the hypocotyl length, and/or the number of dead seeds.

These and other advantages of the present invention will become more apparent from a detailed description of the invention.

**Brief Description of the Drawings**

In the accompanying drawings, which are incorporated in and constitute a part of this specification, embodiments of the invention are illustrated, which, together with a general description of the invention given above, and the detailed description given below serve to example the principles of this invention.

Figure 1 is a schematic block diagram showing various components of the present system and method;

Figure 2 is a perspective view of a lid of a germination box, i.e., a sandwich box, having several soaked pieces of blotter paper placed therein with a plurality of seeds pressed thereon;

Figure 3 is a side view of a sandwich box with the base secured to the lid of Figure 2 and placed in a nearly upright position;

Figure 4 is a perspective view of an inverted scanner enclosure of the present invention with the scan drawer in the closed/scan position;

Figure 5 is a perspective view of the inverted scanner enclosure of Figure 4 with the scan drawer in the open position, with a sandwich box lid with blotter paper and seedlings held in place by the lid retainer;

Figure 6 is a cutaway front view of the inverted scanner enclosure of Figure 4 with the drawer face, front enclosure face, and front lid face removed to show the position of the scanner relative to the enclosure and lid tray;

Figure 7a is a sectional view of the of the inverted scanner enclosure of Figure 4 taken along the line 7—7 in Figure 6, with the drawer face, front enclosure face, and front lid face in place and in the scan drawer in the closed position;

Figure 7b is a sectional view of the of the inverted scanner enclosure of Figure 4 taken along the line 7—7 in Figure 6, with the drawer face, front enclosure face, and front lid face in place and in the scan drawer in the open position;

Figure 8 is an exploded view of the of the inverted scanner enclosure of Figure 4;

Figure 9 is a color screenshot of the seedling analysis software of the present invention;

Figure 10 is a flowchart showing a general overview of the seedling analysis software of the present invention;

Figure 11 is a flowchart showing generally how the seedling analysis software of the present invention determines seedling skeletons from a digital seedling image; and

Figure 12 is a flowchart showing generally how the seedling analysis software of the present invention determines the primary axis for each seedling.

**Detailed Description of the Preferred Embodiment**

"Circuit communication" as used herein indicates a communicative relationship between devices. Direct electrical and optical connections and indirect electrical and optical connections are examples of circuit communication. Two devices are in circuit communication if a signal from one is received by the other, regardless of whether the

signal is modified by some other device. For example, two devices separated by one or more of the following—transformers, optoisolators, digital or analog buffers, analog integrators, other electronic circuitry, fiber optic transceivers, or even satellites—are in circuit communication if a signal from one reaches the other, even though the signal is modified by the intermediate device(s). As a final example, two devices not directly connected to each other, but both capable of interfacing with a third device, e.g., a CPU, are in circuit communication. As used herein, "input" refers to either a signal or a value and "output" refers to either a signal or a value. As used herein, the term "hypocotyl" refers to the portion of a seedling between the cotyledon and the radicle (root or roots) or between the seed coat and the radicle, if a cotyledon is not apparent. As used herein, the hypocotyl starts at the seed coat or cotyledon and ends at the first root hair, which is considered to be the beginning of the seedling radicle. As used herein, the term "overlapped" in the context of overlapped seedlings means seedlings that cross, seedlings that share an edge, and/or seedlings that otherwise form part of a single object in a seedling image (e.g., seedlings that form part of the same object in a binary image determined from an image of seedlings) after the seedling image is processed.

Referring now to Figure 1, an overview of certain elements of a system 10 according to the present invention are shown. In the broadest sense, the system 10 of the present invention comprises a system and method for automatically determining a seed vigor index for a batch of seeds. A particular embodiment of that system is shown in Figure 1. With reference to that figure, the system 10 of the present invention comprises a computer system 12 having a processor unit 14 in circuit communication with one or more display devices, e.g., monitor 16 and/or printer 18, in circuit communication with one or more input devices, e.g., mouse 20 and/or keyboard 22, and in circuit communication with an image capture device to capture an image of one or more seedlings germinated from a representative sample of seeds from the batch of seeds being analyzed. Preferably the image capture device is an inverted scanner 24, i.e., an upside-down scanner, which preferably comprises an ordinary scanner inverted in a special enclosure that holds the scanner in an inverted configuration and holds one or more seedlings proximate to the

glass scanning surface of the scanner for scanning. The seedlings to be analyzed are preferably grown using a novel method of using a germination box 26, i.e., a sandwich box 26, to grow seedlings having a specific orientation of hypocotyl and radicle, as will be explained herein.

According to the present invention, the seedlings under analysis are preferably grown in a sandwich box that is vertical or nearly vertical. In the prior art, seedlings are typically grown in sandwich boxes with the longitudinal axis of the sandwich box positioned horizontally. Seedlings grown in such a manner typically have hypocotyls that grow vertically and radicles that grow horizontally, which is not conducive to automatic analysis of a single scan. According to the present invention, seedlings are grown with their hypocotyls generally parallel to their radicles, or at least grown with their hypocotyls essentially in the same plane as their radicles. This is accomplished by allowing the subject seeds to germinate in the dark in a sandwich box that is vertical or nearly vertical. Under such conditions, the seedlings tend to grow with their hypocotyls generally parallel to their radicles, or at least with their hypocotyls in generally the same plane as their radicles, either of which facilitates analysis using a single scan because the scan can include both seedling radicles and seedling hypocotyls. In the dark, seedling hypocotyls tend to grow upward and seedling radicles tend to grow downward. Placing the sandwich box vertical or nearly vertical allows the dark-germinated seedlings to grow so that they essentially lie flat on the blotter paper.

To further facilitate scanning of seedlings, the seedlings are preferably grown on a suitable medium, e.g., blotter paper, in a very shallow container, e.g., a lid of a sandwich box. This facilitates scanning by permitting the shallow container, e.g., the lid of the sandwich box, to be placed proximate to the scanning surface without handling the medium on which the seedlings were grown. Referring now to Figure 2, a shallow container, i.e., a lid 50 of a sandwich box 26 is shown. The lid 50 preferably has a shallow lip 52. The particular sandwich boxes 26 used in the implementation of this embodiment of the present invention are about 6 3/8" by about 9 5/8" and use blotter papers that are cut to about 5½" by about 9". These boxes are available from numerous sources known to

those in the art, e.g., Model 600C available from Pioneer Packaging, Dixon, KY. These boxes typically have a hinge and a closing clasp, neither of which are shown in Figure 2. Although these boxes are preferred because of their size, virtually any sized germination sandwich box could be used, subject to availability. The blotter paper is preferably blue germination blotter paper available from numerous sources known to those in the art, e.g., Anchor Steel Blue Seed Germination Blotter paper available from Anchor Paper Company, St. Paul, MN. The blue color of this germination paper provides a relatively high contrast with the seedlings and facilitates separating seedlings from the blotter background in the seedling images.

Preferably, two sheets 54, 55 of blotter paper are used to germinate lettuce (*Lactuca sativa* L.) seeds by providing adequate moisture to allow the lettuce seeds to germinate in the germinator. More sheets might be needed to provide adequate moisture for other seeds, e.g., soybeans (*Glycine max* (L.) Merr.). These two sheets 54, 55 are preferably thoroughly wetted with a suitable germination solution, e.g., distilled water. Also shown in Figure 2 are fifty (50) lettuce seeds 58 arranged in two rows of twenty-five (25) seeds, with the centers of the seeds nominally spaced at about 8.8 mm apart, with the two rows nominally spaced 5.0 cm apart. This arrangement of lettuce seeds tends to minimize overlap between seedlings germinated at 3 days. Seeds can be placed in a proper configuration using any suitable method, including using a vacuum plate known to those in the art, with the vacuum plate having two rows of twenty-five holes each spaced as discussed above. The foregoing spacings are preferred for lettuce seeds. Seeds of different species which are to be germinated into seedlings for analysis may require different spacings. An important criterion is that the seeds be spaced so as to tend to minimize overlap between seedlings. For example, on the one hand, one species of impatiens, *Impatiens walleriana*, tends to have a single primary root and several shorter secondary roots, and can be germinated using the 2x25 configuration discussed above for lettuce. On the other hand, seedlings of a different species of impatiens, *Impatiens balsamina*, are bigger than the other species and typically have four to six roots of about the same size that fan out, and consequently, their seeds need to be spaced further apart, e.g., 15 mm apart in

2 rows of 15 seeds each, with the rows being 5.0 cm apart. Some species, because of the relatively large size of their seedlings, would not be germinated on blotters but on a different medium more suitable for larger seedlings, e.g., on 10 x 20 inch paper germination towels. In this situation, the inverted scanner of the present invention might not produce acceptable results and another digital imaging device would be used to capture the appropriate seedling image, such as a digital camera positioned essentially perpendicular to the seedlings, e.g., above the seedlings when positioned horizontally. The seedling analysis software of the present invention would function acceptably well on a seedling image of suitable resolution and quality acquired by virtually any digital imaging device, although some of the software parameters of the present invention might need to be altered to take into account different characteristics of seedling images acquired by different digital imaging devices, e.g., different effective dpi levels.

After the seeds are placed on the two sheets 54, 55 of wet blotter paper, the seeds are preferably gently pressed onto the blotter paper. This pressing depresses the seeds into the surface of the upper piece 55 of blotter paper, which helps prevent the seeds 58 from falling off of the blotter paper when the sandwich box 26 is placed in a vertical or near vertical position in the germinator. A hard plastic plate, e.g., a smaller sandwich box, can be used to gently press the seeds 58 onto the upper sheet 55. Larger seeds can be held in place using the same method, or by sandwiching the seeds between two pieces of moistened germination blotters.

Referring now to Figure 3, after the seeds are pressed onto the upper sheet 55 of blotter paper, the base 60 of the sandwich box 26 is connected to the lid 50 and placed in a germinator (not shown) in a vertical or nearly vertical position. The angle $\alpha$ between the lid 50 and an imaginary vertical surface 62 is preferably as low as practical ($\alpha$=0° indicates a vertical orientation) for the specific type of seed being germinated and analyzed. For example, for lettuce, the angle $\alpha$ is preferably between 0° (vertical) and 15°, more preferably between 0° (vertical) and 10° and most preferably 0° (vertical). As another example, for soybeans, the angle $\alpha$ is preferably between 0° (vertical) and 45°, more preferably between 0° (vertical) and 10° and most preferably 0° (vertical). No matter

which crop being analyzed, it is desirable and preferable to have the angle $\alpha$ be as close to 0 degrees (vertical) as possible to ensure that the seedling roots (radicles) grow down and the seedling shoots (hypocotyls) grow up, so that they are preferably essentially in the same plane, which facilitates the seedling analysis.

After the seedlings have germinated for a number of days, preferably 3-5 days for certain species, more preferably 3 days for certain species, an image of the seedlings must be taken. Although numerous devices can be used to create an image of the seedlings, e.g., a scanner or a digital camera, it is most preferred to use an inverted scanner, i.e., an upside-down scanner, to generate a digital image of the seedlings. In the alternative, a scanner in the vertical position, nearly vertical position, or some other rotated position (preferably rotated at least 90° from the horizontal) can be used to generate a digital image of the seedlings (under the rationale that if the seedlings can be germinated at a vertical or nearly vertical angle, then they can be scanned at a vertical, nearly vertical angle, or some other angle as well). In either case, the specially enclosed scanner of the present invention has been shown to provide uniform lighting across seedling blotters, which leads to relatively uniform intensity within images and relative uniformity of lighting between scans. Digital cameras, in contrast, can have relatively non-uniform lighting within an image (e.g., "hot spots") and can vary significantly in intensity from image to image. Figures 4-8 show an inverted scanner 24 according to the present invention. The inverted scanner 24 comprises a scanner 70 held upside-down in a scanner enclosure 72. A very important criterion for the scanner 70 is that it function inverted, i.e., upside-down. One suitable scanner 70 is a UMAX Astra 2000U scanner, which is commonly available from numerous sources. This scanner and other scanners are typically shipped with a scanner lid (not shown) that would be used to cover the material being scanned during scanning. This lid is removed from the scanner 70 before it is placed in the inverted scanner enclosure 72. The scanner enclosure 72 comprises a scanner lid 74 and a base 76. The scanner lid 74 comprises a lid top 78, front lid face 80, two side lid faces 82, 84, and a rear lid face 86. The four pieces 80, 82, 84, and 86 may be made from sheet metal, bent from extensions of lid top 78, and welded or brazed together at 90° angles to form the lid 74. The scanner 70 is held to the assembled

scanner lid 74 with four sheet metal screws 90a-90d extending through holes in pieces 80 and 86 and into the plastic feet or legs of the scanner 70. The scanner base 76 comprises a base bottom 92, front base face 94, two side base faces 96, 98, and a rear base face 100. The four pieces 94, 96, 98, 100 may be made from sheet metal, bent from extensions of base bottom 92, and welded or brazed together at 90° angles to form the base 76.

The inverted scanner 24 has a scanner drawer 110 comprising a drawer base 112 and a drawer face 114 integrally formed with or physically annexed to (e.g., riveted to, bolted to, welded to, brazed to, adhered to, etc.) drawer base 112. The drawer base preferably has a scanning tray 115, which accepts and generally retains in place the shallow dishes containing the germinated seedlings, e.g., sandwich box lid 50. The tray 115 may be formed from four metal L-brackets 116a-116d which are integrally formed with or physically annexed to (e.g., riveted to, bolted to, welded to, brazed to, adhered to, etc.) drawer base 112. Scanning tray 115 is preferably positioned so that the entire shallow dish (e.g., sandwich box lid 50) is positioned directly below an active scanning area of the scanner 70. The four metal L-brackets 116a-116d are preferably sized so that the shallow dish (e.g., sandwich box lid 50) may be easily placed into the tray 115 for scanning and easily removed therefrom. The front base face 94 of scanner enclosure has an opening 120 therein, through which the drawer base 112 extends. Drawer face 114 has four strips 118a-118d of 1/8" thick rubber about ½" wide secured thereto with suitable adhesive, which form a rectangular light seal, which helps prevent extraneous light from entering the enclosure 72 through opening 120. Drawer face 114 also has a knob or pull 122 used to open and close drawer 110.

Inside enclosure 72 are two shelves, a wider shelf 130 and a narrower shelf 132, that support scanner 70. The shelves 130, 132 are preferably formed from pieces of bent sheet metal and are integrally formed with or physically annexed to (e.g., riveted to, bolted to, welded to, brazed to, adhered to, etc.) the base bottom 92, front base face 94, two side base faces 96, 98, and rear base face 100 in the positions shown in the figures. Anterior surfaces 134, 136 of the shelves 130, 132 have physically annexed thereto (e.g., riveted to, bolted to, welded to, brazed to, adhered to, etc.) and support at least one pair of slide

brackets 140, 142. Slide brackets 140, 142 accept slides 144, 146 physically annexed to (e.g., riveted to, bolted to, welded to, brazed to, adhered to, etc.) scanner drawer 110 in such a manner that scanner drawer 110 can be moved from the open position to place seedlings on the scanner tray 115 (Figures 5a and 5b) and the closed position for scanning (Figure 4). Roller brackets 140, 142 and rollers 144, 146 of scanner drawer 110 are positioned with respect to anterior surfaces 134, 136 and slide base 112 such that the bottom of scanning tray 115 is positioned about 7/16" below the bottom enclosure plane 150 of inverted scanner 70. Since scanner glass 152 of this particular inverted scanner 70 is located about 1/8" into the bottom enclosure plane 150, the scanning tray 115 is positioned about 9/16" below the bottom of scanner glass 152 of this particular inverted scanner 70. Scanner 70 rests upon two 1" strips of ¼" thick rubber, supported by shelves 130, 132, with the bottom edge 164 of scanner enclosure lid 74 resting upon the upper edge 166 (all the way around) of scanner enclosure base 76. Enclosure base 76 also has openings 170a, 170b through which power and data cables of scanner 70 extend to be placed in circuit communication with a power source and a data acquisition system, e.g., computer system 12.

The inside of enclosure 72 is preferably painted, or otherwise coated or anodized, a black color, preferably matte or flat black, to help prevent stray light inside the enclosure 72 from affecting the scanning procedure.

Although the individual parts of scanner enclosure 72 have been described as being made of metal or sheet metal and physically annexed (e.g., riveted, bolted, welded, brazed, adhered, etc.) together, this description was merely illustrative of one suitable construction method for the inverted scanner 24. In the alternative, most or all of the component parts of scanner enclosure 72 may be formed from any suitable plastic or polymer material and secured using any suitable combination of fasteners, adhesives, and/or welding or brazing. Additionally, in the alternative, rather than placing an off-the-shelf enclosed scanner in a special second, inverted enclosure, the internal parts of a scanner can be placed in a single dedicated inverted scanner enclosure (not shown) having

a drawer with a scanning tray that accepts the small containers in which the seedlings are grown (all not shown).

In use, a plurality of seedlings are germinated in accordance with the text accompanying Figures 2 and 3 on wet blotter paper in the lid 50 of a sandwich box placed in a vertical or nearly vertical position. After a desired amount of seedling growth, e.g., after a predetermined number of days of growth, e.g., 3 days, the lid 50 is removed from the base 60 of the sandwich box 26 for scanning. With the scanner in an idle mode, the user pulls drawer pull 122 to withdraw drawer 110 far enough out of enclosure base 76 that the lid 50 may be laid into scanning tray 115. Next, the drawer 110 is pushed back into enclosure base 76 so that the rubber lightly 118a-118d engages front face 94. Next, a scan of the seedlings is taken with inverted scanner 70, preferably using the separate scanner interface/scanning program (not shown) that accompanied the scanner 70. The file representing the scan is then saved to a suitable medium, e.g., a fixed disk of a hard disk drive in computer unit 14, preferably with the interface/scanning program that accompanied the scanner 70. A 24-bit scan at 200 dots per inch (dpi) is sufficient to achieve usable results with lettuce seedlings. For other crops, a higher resolution might be needed or a lower resolution might produce adequate results. The area of the scan is set to match the size of the rectangular blotters 54, 55 on which the seedlings were grown. The scan may/may not include regions outside the area of blotters 54, 55.

The software of the present invention contains seedling analysis software and other functions associated therewith. The software is preferably a stand-alone executable program that executes on systems operating typical operating systems, e.g., Windows 95 and Windows 98. Figure 9 is a screenshot 200 captured from the software of the present invention executed on computer system 12 executing Windows 98 as its operating system. Computer system 12 can be a DELL Dimension XPS T600 computer system having a 600 MHz Pentium III processor with 256 MB of memory, placed in circuit communication with inverted scanner 24 via an appropriate communications protocol, e.g., a Universal Serial Bus (USB) communications protocol in the case of the UMAX Astra 2000U scanner.

The screenshot 200 shows icons implementing some of the functions of the software, such as creating a new seed vigor database, opening a seed vigor database, saving a seed vigor database, showing a seed vigor database, opening a seedling image, saving a seedling image, analyzing a blotter of seedlings, analyzing a lot of seeds, generating a histogram of seedling information, and printing. Actuating the "New Database" icon clears the current database stored in system RAM. A database in the seedling analysis software of the present invention comprises the following elements: seed lot ID, date of testing, date of seed acquisition, vigor index, growth value, uniformity value, individual measurements for each seedling (e.g., hypocotyl length, radicle length, total length, and ratio of hypocotyl length to radicle length), statistical calculations for the particular seedling image (mean hypocotyl length, mean radicle length, and standard deviation of hypocotyl length, radicle length, total length, and ratio of hypocotyl length to radicle length), weights for the seed vigor index calculation, and comments. Actuating the "Open Database" icon opens a dialog box that prompts the user to enter or select the name of an existing database file. When the dialog box is actuated, the contents of the selected file are loaded into memory as the database. When the "Save Database" icon is pressed, the database contents are written to the opened file or to a new file selected by the user. Actuating the "Show Database" icon opens up a window that displays the contents of the database (e.g., seed lot ID, date of testing, date of seed acquisition, vigor index, growth, uniformity, and comments). Actuating the "Open Image" icon opens a dialog box that prompts the user to enter or select the name of a file corresponding to a scanned image of a blotter and seedlings. When that dialog box is actuated, the file corresponding to the scanned image of a blotter and seedlings is loaded into memory and available for analysis. Actuating the "Save Image" icon saves the image being displayed on the screen to a file in a standard graphic file format, such as either the JPEG or the TIFF file format. Actuating the "Analyze Blot" button initiates software analysis, i.e., the routines of Figures 10-12, of the blotter and seedling image currently loaded in memory preferably by actuating the "Open Image" icon. When the software processing is complete for that one image, a seed vigor index and other values are determined from the information determined from that

one image, and the test results are output to the screen (see Fig. 9). Actuating the "Analyze Lot" icon opens a dialog box to have the user select a plurality of seedling image(s) of the particular lot being tested for a combined analysis. When the dialog box is actuated, the software analyzes each image individually, in turn, determines the hypocotyl lengths, radicle lengths, etc. of all the seedlings in each image, determines a seed vigor index and other values using the information about all of the seedlings determined from the multiple images, and the combined test results are output to the screen. The "Analyze Lot" function allows a user to combine analyses of more than one image into a single set of seed vigor values for a specific lot of seeds. For example, the standard number of seeds analyzed in seed vigor testing is 200 seeds; with the 50-seeds-per-blotter configuration described herein for certain species, the user would select four images of four different seedlings and blotters for a total of 200 seeds, and the software of the present invention presents a seed vigor index and other values for all 200 seedlings. The user can select virtually any number of blotters for combined analysis via the "Analyze Lot" function. Actuating the "Histogram" icon opens a window that displays the histogram of a user selected parameter such as hypocotyl length, radicle length, and total length. The "Print" icon prints out the image being displayed on the screen to the printer 18. These and other functions can also be executed via menu commands, as known to those skilled in the art.

Referring now to Figures 10-12, an overview of the seedling analysis software of the present invention is shown. The source code for a majority of the seedling analysis software is appended hereto as Listing 1, Listing 2, and Listing 3, and incorporated herein by reference. A general overview 210 of the seedling analysis software is found in Figure 10, which begins at 212. As indicated at 214, the seedling analysis software requires an image of a blotter and seedlings. This preferably takes the form of a data structure corresponding to the scanned image of a blotter and seedlings that was loaded into memory with the "Open Image" icon/function.

The data structure corresponding to the scanned image of a blotter and seedlings is processed by routine 216 to determine seedling skeletons. The seedling skeletons are preferably a locus of adjacent pixels, one pixel wide, that extends from the top of the

hypocotyl to the bottom of the primary root of the radicle, roughly through the center of the seedling and includes portions for each structure of the seedling, including all root hairs, etc. Figure 11 shows additional details about the skeleton determining routine 216, which starts at 218. At task 220, a seedling image is generated by removing the background, i.e., removing the blotter from the image leaving an image of the seedlings. The seedling image generated by step 220 is preferably a binary image of the seedlings (cotyledons, hypocotyls, radicles, root hairs, etc.) of the same size as the original image with the seedlings (cotyledons, hypocotyls, radicles, root hairs, etc.) in the foreground. This binary image is preferably generated using thresholding based on the intensity of red pixels that occur most frequently in the original image. More specifically, the software at task 220 determines the intensity of red $r_{max}$ (on a scale of 0 to 255 for 24-bit color) that occurs most frequently in the original image. The threshold for the thresholding step is then set to $r_{max}$ plus a certain value, preferably 40 for certain plants (such as lettuce). Next, the software at task 220 generates the binary image with pixels meeting the following criterion being in the foreground: a red intensity of between ($r_{max}$ + 40) and 255, a green intensity of between 0 and 255 (any green intensity), and a blue intensity of between 0 and 255 (any blue intensity). The resulting image is a binary image with the seedlings (cotyledons, hypocotyls, radicles, root hairs, etc.) in the foreground.

Next, at step 222, the binary seedling image generated at step 220 is filtered to remove noise. Under the assumption that noise will take the form of small groups of pixels, this filtering step preferably comprises removing all objects (an object is a contiguous group of pixels) from the image having less than a certain number of pixels. To facilitate the filtering, all the objects in the binary seedling image are preferably labeled and listed in a List of Objects data structure, which includes for each object in the binary seedling image: a unique object label, the bounds of that object ($x_{min}$, $x_{max}$, $y_{min}$, and $y_{max}$), and the number of pixels in that object.

To create the List of Objects, object labeling (also known as connected-component labeling) is used. This is a technique used for extracting connected components (i.e., objects) from a binary image based on a pixel neighborhood system. Typically, a pixel

neighborhood system is either 4-connected or 8-connected. In the 4-connected system, a pixel's neighbors (i.e., pixels that are "connected" to the pixel) are the pixels adjacent at north, south, east, and west of the pixel. In the 8-connected system, a pixel's neighbors include the four pixels mentioned as well as northeast, northwest, southeast, and southwest. The object labeling of step 222 is preferably implemented using the 8-connected system; pixels are connected diagonally as well as vertically and horizontally.

In performing the object labeling of step 222, preferably the pixels in the binary image are scanned in raster-scan order. The first block of contiguous foreground pixels (called a "run") in the same row are marked as belonging to the first object. When another run of foreground pixels is encountered in the same row, then the run is labeled as the next object. The next run encountered in the row is marked as the next object, and so on, until the end of the row. When a run is encountered in the next row of pixels, it is labeled as a new object unless any of its pixels are connected to runs in the row above. If the run is connected to only one run in the upper row, then the same label is used for the run as the one for the run above. If the run is connected to two or more runs, then the label for the run is the first run encountered in the above row, and the labels for the other connected runs are relabeled as equivalent to that label. These equivalent labels are rewritten to the representative label by performing another raster scan. At the end of the whole procedure, each connected component (i.e., each object) has a unique label. For each object, the bounds and size in pixels are determined.

Having generated a List of Objects for the binary seedling image generated at step 220, the filtering process can be done by removing all objects from the list having fewer than a predetermined number of pixels, preferably fewer than 100 pixels for certain plants (such as lettuce scanned at 200 dpi). Next, from this filtered List of Objects, a filtered binary seedling image is created by adding the objects remaining in the filtered List of Objects to a new binary image that represents the filtered binary seedling image.

Then, at step 224, the filtered binary seedling image created in step 222 is smoothed to remove any jagged edges resulting from any of the previous processing steps, which facilitates skeleton determination. The filtered binary seedling image created in step

222 is preferably smoothed using a binary image median filter. A preferred binary image median filter uses a kernel size of three pixels (i.e., a three-by-three square of pixels) and sets a pixel to the same value as the majority value of that pixel and its eight surrounding neighbors. For example, a pixel that is a logical ONE and having four or more surrounding neighbors that are also a logical ONE (for a total of five or more ONEs) would be set to a logical ONE, and a pixel that is a logical ONE and having three or fewer surrounding neighbors that are also a logical ONE (for a total of four or fewer ONEs) would be set to a logical ZERO. The resulting image is a smoothed, filtered binary image. The particular median filter used was adapted from R. Gonzalez and R. Woods, Digital Image Processing, 2nd ed., (1992).

Next, at step 226, the smoothed, filtered binary image generated at step 224 is thinned to produce the seedling skeletons. Preferably, each object in the smoothed, filtered binary image is iteratively made thinner and thinner until the object is reduced to a number of line segments that are one pixel in width. In the broader sense, this step is a form of medial axis transformation. The particular thinning algorithm used was adapted from N. Yagi, S. Inoue, M. Hayashi, E. Nakasu, K. Mitani, M. Okui, S. Suzuki, Y. Kanatsugu, C gengo de manabu jissen gazou shori [Learn Image Processing in C], Ohm-sha [publisher], pp. 53-54, 1997, and is set forth in Listing 3. The result is a thinned, smoothed, filtered binary seedling image that represents the skeletons for the seedlings in the original image. Next, object labeling as discussed above is performed on the resulting thinned, smoothed, filtered binary image, which results in a List of Objects data structure for the skeletons in the thinned, smoothed, filtered binary seedling image, i.e., a List of Seedling Skeletons, which includes for each seedling skeleton: a unique object label, the bounds of that skeleton ($x_{min}$, $x_{max}$, $y_{min}$, and $y_{max}$), and the number of pixels in that skeleton. Finally, the List of Seedling Skeletons is filtered by removing all entries having fewer than a predetermined number of pixels, e.g., 15 pixels, again to eliminate any noise generated by any of the foregoing steps. This filtering action is performed in the same manner as step 222 on the List of Seedling Skeletons. At 228, program control returns to the flowchart of Figure 10.

Referring back to Figure 10, next at routine 230 the seedling analysis software of the present invention determines the primary axis for each seedling. At this point in the analysis, a filtered List of Seedling Skeletons has been generated; however, each seedling skeleton does not necessarily consist of a single seedling. If any of the seedlings were overlapping or very close to each other in the original scanned image, the corresponding seedling skeleton will consist of those multiple seedlings. Thus, the software preferably comprises a routine for separating multiple seedlings and more preferably comprises a routine for separating multiple seedlings using simulated annealing. Next, in general, the software determines the hypocotyl/radicle separation point for each single or separated seedling, at 232. Then, at 234, the software determines the lengths and other statistics used to determine the seed vigor index and calculates the seed vigor index. Next, the software determines a graphical overlay to be displayed with the original scanned image to show the locations of the hypocotyls and radicles of the scanned seedlings, at 236. Figure 12 provides additional information about the routines 230, 232, and 234. Finally, at task 238, the software displays the vigor index, other statistical information, and the graphical overlay combined with the original scanned image.

Referring now to Figure 12, additional information about the seedling analysis is shown, starting at 240. The routines of Figure 12 are performed for each seedling skeleton listed in the filtered List of Seedling Skeletons. First, at 242, a junction graph for the skeleton is generated. In general, a junction graph is a data structure comprising a list of types of junctions in the skeleton, among other information. Each pixel in the skeleton is classified as either a terminal junction (an endpoint of the skeleton), or a connector junction (presumably where root hairs or seminal roots extend from the primary root, or where multiple seedlings overlap or meet), or not a junction. To define the junctions, a new neighborhood system was developed and used in the software of the present invention. The new neighborhood system uses a three-by-three neighborhood, and places primary emphasis on the N, S, E, and W positions and lesser emphasis on the NW, NE, SW, and SE positions of a standard three-by-three neighborhood map. More specifically, with respect to a pixel (i.e., a logical ONE) in the center of the three-by-three

neighborhood, if there is a pixel (i.e., a logical ONE) at any of the N, S, E, or W positions, then they are considered to be neighbors of the center pixel. A pixel at the NE corner is only considered to be a neighbor of the center pixel if there is no pixel (i.e., no logical ONE) at either the N or the E positions. Similarly, a pixel at the NW corner is only considered to be a neighbor of the center pixel if there is no pixel (i.e., no logical ONE) at either the N or the W positions. The same applies to the SW and SE positions: pixel at either the SE corner or the SW corner is only considered to be a neighbor of the center pixel if there is no pixel (i.e., no logical ONE) at either the S or the E positions, or at either the S or the W positions, respectively. This new neighborhood system facilitates the generation of a junction graph used in primary axis generation and seedling separation. Using this new neighborhood system, a pixel is considered to be a junction if and only if the number of its neighbors is not two. If a pixel has only one neighbor under the new neighborhood system, then it is classified as a terminal junction. If a pixel has three or more neighbors under the new system, then it is classified as a connector junction. The remaining pixels (those with two neighbors under the new neighborhood system) form the "edges" of the junction graph, which are linear curve segments (portions of the skeleton) between junctions. An edge, i.e., a linear curve segment, cannot contain any junctions in this particular junction graph data structure.

The junction graph is preferably determined as follows. Within the bounding box of the seedling skeleton, all pixels belonging to the skeleton are tested for the number of neighbors. A pixel that does not have exactly two neighbors is marked as a junction, and information about where its neighbors are (e.g., NW and E) is stored. When the seedling skeleton is completely scanned (i.e., all pixels are processed), we have all the junctions (nodes) of the graph for that particular skeleton. As discussed above, an edge formed by two connected junctions is made up of all pixels connecting them. Pseudocode for finding the edges for the junction graph is as follows:

```
for each junction j
    for each neighbor n of junction j
        while current pixel is not a junction
            current pixel := next pixel in the curve segment;
        end while;
        k := junction id of the current pixel;
```

```
                if edge (j, k) has not been inserted
                    insert (j, k) into junction graph;
                else
                    if length of this curve segment < edge (j,k)
                        remove (j, k);
                        insert (j, k) with new curve segment distance;
                    end if;
            end if;
            end for;
        end for;
```

For each edge, the length and the angles it makes at its junctions with respect to the x-axis are computed. The angle for each junction is computed by computing the line formed from the junction and the fifth pixel in the curve segment from that junction. If there are less than five pixels in the curve segment, the angle is computed from the line formed from the junction and the other junction in the segment. The angles are used as a criterion for separating multiple seedlings. The length of each edge is calculated in pixel lengths, with one pixel length being added for pixels next to or on top of (N, S, E, or W to) the next pixel and $\sqrt{2}$ pixel lengths (approximately 1.414) being added for pixels at a diagonal to (NW, NE, SW, or SE to) the next pixel. The edge lengths are used to determine the primary axis of the skeleton, as a criterion for separating multiple seedlings, and ultimately to determine the hypocotyl length, the radicle length, and the total seedling length.

Next, at task 244, the seedling analysis software of the present invention determines whether the skeleton consists of a single seedling or multiple seedlings. This preferably makes use of information about seed coats, cotyledons, or both seed coats and cotyledons. Preferably, information about seed coats and/or cotyledons is determined using a thresholding function similar to task 220 when the binary seedling image was generated. More specific to seed coats and cotyledons, the software performs two thresholding operations on the images, one with parameters directed toward seed coats and the other with parameters directed toward cotyledons. For specific plants, e.g., lettuce, the binary seed coat image is preferably generated with pixels meeting the following criterion being in the foreground: a red intensity of between 60 and 255, a green intensity of between 0 and 255 (any green intensity), and a blue intensity of between 0 and 100. For specific plants, e.g., lettuce, the binary cotyledon image is preferably generated with pixels

meeting the following criterion being in the foreground: a red intensity of between 200 and 255, a green intensity of between 200 and 255, and a blue intensity of between 0 and 200. These two binary images are then labeled using object labeling to generate a List of Objects data structure for each binary image, which are then filtered using the filter routine 222 to filter out objects less than a certain pixel size, e.g., less than 10 pixels. The resulting binary seed coat image and binary cotyledon image have objects corresponding to seed coat objects and cotyledon objects, respectively, in the original scanned image. These seed coat objects and cotyledon images can be used to determine how many seedlings are present in each skeleton.

Based on the assumption that there is one cotyledon object (actually a pair of cotyledons, which appear as a single cotyledon object in the binary cotyledon image) and exactly one seed coat object for each seedling, the number of cotyledons/seed coats indicates the number of seedlings on the skeleton. However, in some seedlings only a cotyledon object is present (e.g., because the seed coat fell away from the cotyledon), in some seedlings only a seed coat is present (e.g., because the cotyledon has not emerged or remains covered by the seed coat), and in some seedlings both a seed coat and a cotyledon are present (e.g., because the cotyledon has partially emerged from the seed coat). Accordingly, sometimes a seed coat object and a cotyledon object are merged into a single object to indicate a cotyledon for a single seedling if they are close enough (e.g., within 10 pixels). In the present implementation of the present invention, a List of Objects is created for each of the cotyledon binary image and the seed coat binary image. Not all the objects in the List of Objects representing seed coats, cotyledons, or seed coats and cotyledons are associated with the particular skeleton being analyzed. Bounding boxes expanded by a number of pixels, e.g., eight pixels, in all four directions (e.g., $x_{min}$-8, $x_{max}$+8, $y_{min}$-8, and $y_{max}$+8) are used to associate the seed coat/cotyledon objects with each skeleton. That is, only seed coat/cotyledon objects in the Lists of Objects located completely within or partially within the expanded bounding box of that skeleton are associated with that skeleton.

Next, for each seed coat/cotyledon object in the Lists of Objects for that skeleton, the software determines the junction closest to the object. More specifically, for each cotyledon object, the junction closest to the center of the object is found (such a junction is called a cotyledon junction), where the center is defined as the center of the bounding box for the object. For each seed coat object, the junction closest to the center of the object that is within 10 pixels is found (such a junction is called a seed coat junction); if no such junction exists, then the seed coat object is discarded from further processing. For each seed coat junction, the existence of a nearby (e.g., within 10 pixels) cotyledon junction is checked. If there is a nearby cotyledon junction, the seed coat junction is discarded. This has the effect of marking a seed coat/cotyledon object as one cotyledon object when the cotyledon is only partially visible from the seed coat. After the above task is completed, the number of cotyledon junctions and the number of remaining seed coat junctions are added to represent the number of seedlings present in the seedling blob. There can be four cases: (1) the sum is zero, (2) the number of cotyledon junctions is one, but the number of seed coat junctions is zero, (3) the number of cotyledon junctions is zero, but the number of seed coat junctions is one, (4) the sum is two or greater. Case (1) represents the case where the seedling is missing a seed coat/cotyledon object, or the size/color of the object was rare that the thresholding did not pick it up. In this case it is assumed that there exists exactly one seedling in the seedling skeleton. Case (2) represents the case where the cotyledon is not covered by a seed coat. In this case, exactly one seedling is assumed in the seedling skeleton. Case (3) represents the case where the cotyledon is entirely covered by a seed coat. It is assumed that the seedling skeleton contains exactly one seedling in this case. Case (4) represents a case where there are multiple seedlings in the seedling skeleton.

The following annotated pseudocode presents additional information on the determination of start junctions in each seedling skeleton (which is determined by task 226), found in Listing 1:

`cotimg` contains the binary image of cotyledons, which was obtained by thresholding the original color image of the blotter and seedlings.

**coatimg** contains the binary image of seed coats, which was obtained by thresholding the original color image of the blotter and seedlings.

**minx** is the minimum x coordinate of the bounding box of the seedling skeleton.

**maxx** is the maximum x coordinate of the bounding box of the seedling skeleton.

**miny** is the minimum y coordinate of the bounding box of the seedling skeleton.

**maxy** is the maximum y coordinate of the bounding box of the seedling skeleton.

**cotblobimg** is **cotimg** cropped by the box (minx - 8, miny - 8, maxx + 8, maxy + 8).

**coatblobimg** is **coatimg** cropped by the box (minx - 8, miny - 8, maxx + 8, maxy + 8).

**cotinfo** is the List of Objects extracted from **cotblobimg** that are larger than or equal to **c_mincotsize** in terms of the number of pixels.

**coatinfo** is the List of Objects extracted from **coatblobimg** that are larger than or equal to **c_mincoatsize** in terms of the number of pixels.

The software examines each seed coat to determine if that seed coat should be associated with a junction:

```
for each object in coatinfo
     (coatx, coaty) = center of the bounding box for the object;
     the ID of the junction in coatinfo that is closest to (coatx, coaty) and within 10 pixels distance from (coatx, coaty) is added to coatindx;
end for;
```

The software then examines each cotyledon to determine its associated junction:

```
for each object in cotinfo
     (cotx, coty) = center of the bounding box for the object;
     the ID of the junction that is closest to (cotx, coty) is added to cotindx;
end for;
```

The software then reconciles the junctions associated with seed coats and/or cotyledons:

```
for each junction ID in cotindx
    cotjunction = junction identified by the junction ID in
cotindx;
    for each junction ID in coatindx
        coatjunction = junction identified by the junction
ID in coatindx;
        if the distance between cotjunction and
coatjunction is less than 10
            the ID of the coatjunction is removed from
coatindx;
        end if;
    end for;
end for;
```

After the above steps, **cotindx** contains the junction ID's of the detected cotyledons, and **coatindx** contains the junction ID's of the detected seed coats. When a seed coat junction is close to a cotyledon junction, e.g., within 10 pixels, the seed coat junction is disregarded (i.e., absorbed into the cotyledon junction). The number of detected cotyledons and seed coats in the seedling skeleton can then be separated into the following four cases:

Case 1: If no cotyledon or seed coat was detected, assume the seedling skeleton contains exactly one seedling.

Case 2: If one cotyledon was detected but no seed coat was detected, assume the seedling skeleton contains exactly one seedling.

Case 3: If one seed coat was detected but no cotyledon was detected, assume the seedling skeleton contains exactly one seedling.

Case 4: If the lists of seed coat junctions and cotyledon junctions for this seedling skeleton does not fit either case 1, case 2, or 3, (which can be checked by adding the number of ID's in **cotindx** and the number of ID's in **coatindx**, and see if the sum is

greater than one) then the software assumes that the seedling skeleton contains multiple seedlings (more specifically, the number of seedlings is determined to be the sum of the number of ID's in cotindx and the number of ID's in coatindx).

The software now has determined enough about the seedling skeleton to determine the result of the branch task at 244, whether the seedling skeleton represents a single seedling or multiple seedlings. On the one hand, if the seedling skeleton was determined to be either Case 1, 2, or 3, then the skeleton is determined to be comprised of only a single seedling. Consequently, the response to the question "Single Seedling?" at branch task 244 in Figure 12 is "Yes" and the code branches to task 246 to analyze the skeleton as a single seedling. On the other hand, if the seedling skeleton was determined to be Case 4,, then that skeleton is determined to be comprised of multiple seedlings. Consequently, the response to the question "Single Seedling?" at branch task 244 in Figure 12 is "No" and the code branches to task 248 to analyze the skeleton as a multiple seedling. In either case, the next step is to determine the "primary axis" of each seedling, i.e., the path from the start junction down to the terminal junction of the primary (longest) root of the radicle. A single routine could be used to determine the primary axis of skeletons representing single seedlings and skeletons representing multiple seedlings. However, the branch at task 244 allows the primary path detection routine for single seedlings to be optimized.

Thus, if the skeleton represents a single seedling, the routine at 246 determines the primary axis of that single seedling. In the single seedling case, there are three subcases discussed earlier: (1) there are no cotyledon nor seed coat junctions, (2) there is one cotyledon junction but no seed coat junction, (3) there is one seed coat junction but no cotyledon junction. For Case 1, the shortest path is computed for each terminal junction to every other terminal junction to detect the primary axis of the seedling. The longest path is considered as the primary path. Since there is no information about which end junction of the primary path is the start junction (i.e., the junction that belongs to the cotyledon), whichever end junction that appears higher in the image (i.e., has a lower y coordinate, assuming the first, top scanline of the screen is y=0, the next scanline is y=1, etc.) is considered to be the start junction. For Case 2, the shortest path is computed from the

cotyledon junction to every other junction and the longest path is considered to be the primary axis. The start junction is the cotyledon junction. For Case 3, the primary axis is found by computing the shortest path from the seed coat junction to every other terminal junction. The longest path is considered to be the primary path. The start junction is the seed coat junction.

This is based on the assumption that the seedling starts from a cotyledon and extends down to the tip of a single primary root. In the alternative, for plants having a plurality of primary, secondary or seminal roots, the primary path routines of the present invention will need to determine a plurality of primary paths, e.g., by selecting the n longer shortest-paths or by determining a statistical value from all the calculated shortest-paths and selecting a number of longer shortest-paths above a determined value based on the statistical value of the shortest-paths. Referring back to the present implementation, given that the start junction is constrained as the cotyledon, the end point is the terminal junction farthest (in terms of path length) from the start junction. For the start junction and all the other terminal junctions (i.e., candidates for the end junction), there can be many, often infinite, ways to traverse the edges (paths). Thus, for each start junction/terminal junction pair, the shortest path is taken. Each junction has a unique identifier associated therewith. Each path is represented by a series of junction identifiers. To find the shortest path from one node to every other node in a junction graph, a routine based on an algorithm presented in E. W. Dijkstra, "A Note on Two Problems in Connection with Graphs," Numerische Math, (1): 269-271 (1959) was used. Preferably, the longest of the shortest-paths is taken as the primary path, i.e., the primary axis of the seedling. The disadvantage of this implementation is that the primary axis for the seedling will not be correct when the seedling makes a large loop, because the shortest path will exclude the length of the loop. In the alternative, the software can detect such loops and correct the shortest path accordingly.

Having determined the primary path, i.e., the primary axis, of the seedling skeleton, the software next determines the hypocotyl/radicle separation point for the seedling skeleton, at 232 (Figure 12). The hypocotyl/radicle separation point is assumed to

be located at the upper most point in the seedling where the root hairs (or secondary roots) extend from the seedling. In terms of the junction graph of the seedling skeleton, the hypocotyl/radicle separation point is assumed to be located at the junction closest to the start junction (cotyledon terminal junction) that does not result in a hypocotyl:radicle ratio (length of hypocotyl divided by length of radicle) that is too low, preferably set at 0.15. Thus, the routine at 232 selects a candidate junction, i.e., the junction in the primary path closest to the start junction, and calculates the lengths of the hypocotyl and radicle based on the assumption that that junction is the hypocotyl/radicle separation point (using the edge lengths in the junction graph). If the ratio of hypocotyl length to radicle length is 0.15 or greater, then that junction is the separation point. If the ratio of hypocotyl length to radicle length is less than the value, e.g., 0.15, the software selects the next junction in the primary path of the skeleton and calculates the lengths of the hypocotyl and radicle based on the assumption that that junction is the hypocotyl/radicle separation point (using the edge lengths in the junction graph). Again, the ratio is determined and compared to the test value, e.g., 0.15, and either results in the determination of a proper separation point or the process repeats itself with the next junction in the primary path of the skeleton until a proper separation point is found (i.e., one that causes the length ratio to pass the test). Once a proper separation point is determined, the lengths of the hypocotyl and radicle are added together to calculate the total length of the seedling skeleton, as indicated at 250, and then the program at 252 either branches back up to perform routine 230 (Figure 12) again for the next seedling skeleton or returns to the code of Figure 11.

Referring back to task 248, if the seedling skeleton was determined to comprise multiple seedlings at task 244, the software must account for the multiple seedling nature of that skeleton. Preferably, the software performs seedling separation, i.e., separates the multiple seedlings into a plurality of single seedlings, which can then be analyzed to determine their separation points, hypocotyl lengths, radicle lengths, and/or total lengths, etc. Preferably, the separation of seedlings in skeletons representing multiple seedlings is done by a routine comprising simulated annealing. A routine based in part on a separation algorithm by H. Ni and S. Gunasekaran, "A Computer Vision Method for Determining

Length of Cheese Shreds," Artificial Intelligence Review, 12: 27-37 (1998) and a simulated annealing routine by S. Kirkpatrick, C. D. Gelatt, M. P. Vecchi, "Optimization by simulated annealing," Science 200(4598): 671-680 (1983) was used.

Recall that from the routine at 244 (or Case 4 in the preceding pseudocode), the software has determined the number and location of the start junction (cotyledon or seed coat junction) for each of the multiple seedlings in the seedling skeleton. The software must next determine the primary path for each start junction, i.e., the primary path for each of the multiple seedlings. This is done by determining a number of possible paths and evaluating them. Because each start junction is connected to one or more other junctions, one of the other junctions can be selected and considered an extension of the seedling. By choosing another junction, then another, a sequence of junctions can be chosen, which in turn defines a sequence of linear curve segments (edges). Any sequence of linear curve segments created this way can be considered a candidate for the structure of the seedling. Since this routine is directed at a junction graph representing multiple seedlings, such a sequence of linear curve segments is created for each start junction, i.e., for each seedling. Each possible combination of all seedling structures (a candidate path for each and every start junction) is called a "configuration." Then, the seedling separation routine can evaluate each configuration of seedling paths and select the configuration determined to be the configuration that has correctly identified the paths of each seedling. Preferably, the seedling separation procedure is implemented as an energy minimization routine, with every configuration having a calculated energy value associated therewith, and the configuration having the lowest energy (or in the alternative the highest energy) is considered to be the correct configuration. That is, the "energy" is defined as a function of the configuration in such a way that a more desirable configuration has lower energy (or higher energy) than a less desirable configuration. The proper configuration is identified by an energy that is minimal as compared to all other configurations. In implementing this energy function and analysis, it is possible to create every possible configuration, compute the energy for each and every configuration, and search for the configuration that minimizes the energy. This brute force method can be accomplished in a reasonable

computation time if the number of junctions and edges is relatively small, and/or if the processing capability of the computer unit 14 is relatively fast. However, the routine does not scale very well, because the search space is exponential, i.e., as the number of junctions and edges increases, the computation time to search the best configuration increases exponentially. Thus, preferably, a stochastic minimization method is used. More preferably, simulated annealing is used. Simulated annealing does not guarantee convergence to a global minimum (i.e., the most desirable configuration of all configurations), but simulated annealing does reach a very low energy configuration with high probability, i.e., simulated annealing identifies a seedling path configuration that very likely contains a correct or very close to being a correct set of paths for each seedling.

The preferred energy function used for this procedure is set forth as follows:

$$SE(s_i) = w_{angle} \sum_{t \in T_{s_i}} angle(t)^2 + w_{separation} I_{separation}(s_i)$$

$$CE(S,U) = \sum_{s_i \in S} SE(s_i) + w_{unused} \sum_{e \in U} length(e)$$

where $SE$ is the energy for seedling $s_i$, $Ts_i$ is the set of pair of junctions $s_i$ on which a turn was made (ignoring turns on edges that are too short), $I_{separation}$ is an indicator function that evaluates to 1 when a hypocotyl/radicle separation exists for $s_i$ and 0 otherwise, $S$ is the set of all seedlings in the seedling blob, $U$ is the set of unused edges (i.e., edges not occupied by any seedlings), $CE$ is the energy for a configuration with $S$ and $U$, and $w_{angle}$, $w_{separation}$, and $w_{unused}$ are constants. It was experimentally found that the following parameters yield satisfactory results: maximum number of annealing loop iterations = 50,000, $w_{angle} = 200$, $w_{separation}=200$, $w_{unused} = 10$. Other values and weights may provide satisfactory results and these values and weights (and other constants used herein) are preferably modifiable by the user to take into account user preference, analysis of different species, etc.

The simulated annealing routine starts with an initial configuration, preferably just the start junction for each seedling. A random change to the configuration is proposed,

preferably a junction is added to one of the seedling paths, and the change is accepted immediately and that junction becomes part of the seedling path if the proposed configuration has a lower energy than the current configuration does. If the proposed configuration has a higher energy than the current configuration, then the proposed configuration has a chance of being accepted or rejected. In the case where the proposal is rejected, then the current configuration remains unchanged. This process of configuration proposal, energy computation, and acceptance/rejection is repeatedly performed. After a number of iterations, a configuration in which the energy is very low should be reached.

Thus, the ultimate goal of simulated annealing is to achieve a configuration that minimizes the energy in an attempt to find the most desirable configuration, without having to evaluate every possible configuration for that seedling skeleton. The formulation for the energy is a primary determining factor of desirable configurations in a problem setting. The "temperature" controls the probability that a change to the current configuration should be accepted. When the temperature is high, the probability of going to a higher-energy configuration can occur as frequently as going to a lower-energy configuration (thus very random); while at a low temperature, changes occur only if it decreases the energy (always downhill).

More specific to the present implementation of the present invention, to define the seedling configuration energy, the following energy determination rules and heuristics are used.

*Rules:*

(1) A primary axis is a polyline (i.e., no branching is allowed).

(2) A seedling cannot reuse one of its edges.

(3) Multiple seedlings can share the same edge in the junction graph.

(4) Edges can be left unused.

*Heuristics:*

(1) Primary axes do not make unnaturally sharp turns.

(2) Edges in the junction graph should be used as much as possible.

(3) It is desirable that all primary axes have a hypocotyl/radicle separation.

Each seedling has a start junction and an end junction, which is the current end of the path for that particular seedling in the skeleton. Each proposed path has a fixed start junction and a varying end junction. Recall that the initial configuration preferably consists of only the start junctions (cotyledon terminal junctions) for the skeleton. Consequently, the start junctions initially are also the end junctions for each seedling. The following loop is performed to iteratively update the proposed seedling configuration:

(a) Among all seedlings in the skeleton, randomly choose one seedling to update.

(b) Determine the neighbor junctions of the end junction using the junction graph for that seedling.

(c) Randomly choose a neighbor junction for that end junction.

(d) If the chosen junction is already in the edge set for the seedling, then attempt to remove the junction from the seedling.

(e) If the chosen junction is anything else, then attempt to add the junction to the seedling.

(f) Decrease temperature and repeat until the maximum number of iterations is reached.

Because junction computation is done at the pixel level, there may be junctions that are close together, which can cause a problem in turn angle calculation. For example, the angle of a turn created by three pixels, a pixel and its N and SE neighbors would be 135°; however, the seedling might not have made any significant turn at all at that point. To avoid this potential problem, the turns of edges that are not sufficiently long are ignored. In this implementation, preferably the turns on edges less than 10 pixels long are ignored.

By the time the maximum number of iterations of the configuration update loop have been performed, the configuration should have the correct primary path for each seedling. A maximum number of iterations of 50,000 was found to yield satisfactory results. Each of the primary paths determined by the routine at task 248 is then processed by software routines 232 and 250 to determine the hypocotyl length, radicle length, and total length.

With that introduction, the following annotated pseudocode describes the seedling separation software, i.e., function **SeparateSeedlings** in Listing 1, in further detail:

**DATA STRUCTURES**

**paths** is a list of primary paths for all seedlings. Because each primary path is represented by a list of junction identifiers, **paths** is a list of lists. Initially the path for each seedling contains only the start junction for that seedling.

**endAngle** is a list whose i'th element is a list of the angles formed by seedling i. Angles of edges are preferably measured with respect to the x-axis (with respect to the horizontal), so the angles between edges are calculated by subtracting the angle of edge from the angle of another. Thus, two parallel edges will have an angle of 0° between them. This data structure does not store angles formed by turning onto edges that are shorter than minEdgeLength.

**pathlength** is a list whose i'th element represents the total length of the present path of seedling i.

**RHseparation** is a list whose i'th element represents the junction identifier of the hypocotyl/radicle separation mark for seedling i.

**RHseparation2** is a list whose i'th element represents the junction identifier of the hypocotyl/radicle separation end. There are several differences between the junctions identified by **RHseparation** and **RHseparation2**. The junction identified by **RHseparation** is the first junction in a seedling's path that is at least 20 pixels away from the start junction. The junction identified by **RHseparation2** is the first junction in a seedling's path encountered after **RHseparation** that is at least 20 pixels away from **RHseparation**. If the first junction from is closer than 20 pixels from **RHseparation**, the software looks at consecutive junctions until the first junction that is at least 20 pixels from **RHseparation** is found and assigns that junction to **RHseparation2**. Unless and until the software has determined both the **RHseparation** and **RHseparation2**, the software does not deem a seedling as having a

hypocotyl/radicle separation point, which is taken into account in the energy function.

`separationlen` is the distance in pixels between the junctions identified by `RHseparation` and `RHseparation2`.

`edgeOccupation` is a list whose i'th element is a set of edges that are occupied by the i'th seedling.

**COMPUTING ENERGY FOR INITIAL CONFIGURATION**

The initial energy is set to the sum of the unused edge penalties for all edges, since no edges are occupied by any seedlings, and the penalty for not having determined any hypocotyl/radicle separation points yet:

```
energy = 0;
for each edge in junction graph
  energy = energy + unusedW * edge_length;
end for;
energy = energy + separationW * number_of_seedlings;
```

where `unusedW` is the weight (scalar) for unused edges, `edge_length` is the length of each respective edge, `separationW` is the weight for each seedling for which the hypocotyl/radicle separation point has not been determined, and `number_of_seedlings` is the number of seedlings in the skeleton being analyzed.

**SIMULATED ANNEALING LOOP**

With this initial set-up, the following loop is performed `loopmax` number of times:

`seedID` is set to the ID of the seedling that is randomly chosen to be updated.

`endjunc` is set to current end junction of the seedling `seedID`.

`neighbors` is the set of ID's all neighboring junctions to `endjunc`. The `neighbors` set includes the previous junction in the path, i.e., the junction in the path before `endjunc`.

`choice` is the ID of a junction randomly chosen from `neighbors`.

`changeAngle` is set to `false`

The particular junction randomly chosen to be `choice` greatly affects the execution of the code, in the sense that the junction chosen by `choice` can either extend the seedling path by one more junction or shorten the path by one junction. A different routine is executed if the *previous* junction in the path is randomly chosen as compared to whether a *subsequent* junction is chosen.

1. `choice` is the same junction as the preceding junction to `endjunc`

If `choice` is the same junction as the preceding junction to `endjunc`, i.e., the software is backtracking back up the seedling path (shortening the path by one junction), the software determines the energy benefit or energy penalty of removing the last junction and edge from the path for the current seedling. Thus, the software computes the change in configuration energy (`deltaEnergy`) if the last junction/edge were to be removed from the primary path of that seedling:

```
deltaEnergy = 0;
if removing the junction/edge results in hypocotyl/radicle
separation to be removed, i.e., the point being removed is
RHseparation2 meaning that removing this junction causes a
hypocotyl/radicle separation point to be lost, for which there
is a penalty
    deltaEnergy = deltaEnergy + separationW;
end if;
```

If removing this junction causes an angle to be changed and/or removed, there may be an energy change in response thereto:

```
if edge_length > minEdgeLength, i.e., if the length of the
edge being removed is less than a minimum edge length (e.g., 10
pixels), which allows us to preferably ignore the small edge
because it might otherwise lead to erroneous angles being used
in the energy function;
    if there are more than one element in endAngle for this
seedling
        angle = angle formed by endjunc, choice, and the
preceding junction to choice;
```

```
            deltaEnergy = deltaEnergy - angleW * angle^2;
        end if;
        set changeAngle to true to indicate that removing this
edge erases an angle;
    end if;
```

If removing this junction causes an edge to become an unused edge, i.e., this seedling is no longer using the edge and no other seedling is currently using this edge, there may be an energy change in response thereto:

```
    if the edge is no longer occupied after removal (no
seedling contains the edge in their primary paths)
        deltaEnergy = deltaEnergy + unusedW * edge_length;
    end if;
```

After determining the change in energy by the proposed removal of this junction, the change in energy is tested to determine if removing this junction was either (1) a change for the better, i.e., the change in energy was negative (`deltaEnergy < 0`), or (2) there was a positive increase in energy (removing this junction was a change for the worse). There is a chance that the change will be made anyway if the condition `(exp(-deltaEnergy / (temperatureConst * temperature)) > random_number_between(0,1))` is satisfied, despite the fact that removing this junction would cause an increase in energy (which allows the routine to find the lowest energy configuration and not merely a local energy minimum). Such a random change is likely to occur when the temperature is high. The probability of this last expression being true (for any value of `deltaEnergy`) decreases with each iteration of the loop (because `temperature` decreases with every loop). If either of these conditions are true, the junction and edge are removed from the seedling path, the energy for the configuration is updated, the length of the seedling path is adjusted, and possibly an angle and/or the separation points, etc. are removed:

```
    if deltaEnergy < 0 or exp(-deltaEnergy / (temperatureConst
* temperature)) > random_number_between(0,1)
        mark that choice no longer occupies the edge;
```

```
            remove choice from the primary axis of the seedling;
            if changeAngle is true
                  remove the last element from endAngle;
            end if;
            if choice is the hypocotyl/radicle separation mark
(RHseparation) for the current seedling
                  remove separation mark from the seedling;
            end if;
            if choice is the separation end point (RHseparation2)
for the current seedling
                  remove separation end point from the seedling;
            end if;
            subtract the edge length from the seedling path length;
            energy = energy + deltaEnergy;
      end if;
```

2. **choice** is any other junction than the preceding junction to **endjunc**

If **choice** is any other junction than the preceding junction to **endjunc**, i.e., the software is adding a new junction to the seedling path (increasing the path by one junction), the software determines the energy benefit or energy penalty of adding this new junction and edge to the path for the current seedling. Thus, the software computes the change in configuration energy (**deltaEnergy**) if this new junction/edge were to be added to the primary path of that seedling:

```
deltaEnergy = 0
```

The software computes the change in configuration energy (**deltaEnergy**) if the edge were to be added to the primary path:

```
if the edge is not already occupied by any other seedling
    deltaEnergy = deltaEnergy - unusedW * edgeLength;
end if;
```

Next, the software computes the change in configuration energy (deltaEnergy) associated with any identification of hypocotyl/radicle separation points or end points by the addition of this junction:

```
    if the current seedling does not already have a
hypocotyl/radicle separation mark (RHseparation)
        if choice has more than 2 neighbors and the current
seedling length is greater than 20
            newseparation = true;
        end if;
    elseif the current seedling has a separation mark (has
RHseparation) but the separation is not complete (does not have
RHseparation2)
   and if the length from the mark to choice is greater than 20,
i.e., choice can be RHseparation2
        deltaEnergy = deltaEnergy - separationW;
        separationcomplete = true;
   endif
```

Next, the software computes the change in configuration energy (deltaEnergy) associated with any change in angle caused by the addition of this junction:

```
    if edge_length > minEdgeLength i.e., if the length of the
edge being removed is less than a minimum edge length (e.g., 10
pixels), which allows us to preferably ignore the small edge
because it might otherwise lead to erroneous angles being used
in the energy function;
        angle = angle formed by preceding junction to end junc,
endjunc, choice;
        deltaEnergy += angleW * angle^2;
        set changeAngle to true to indicate adding this edge
will create an angle;
    endif;
```

After determining the change in energy by the proposed addition of this junction, the change in energy is tested to determine if adding this junction to the path of this

seedling was either (1) a change for the better, i.e., the change in energy was negative (`deltaEnergy < 0`), or (2) there was a positive increase in energy (adding this junction was a change for the worse). There is a chance that the change will be made anyway if the condition (`exp(-deltaEnergy / (temperatureConst * temperature)) > random_number_between(0,1)`) is satisfied, despite the fact that adding this junction would cause an increase in energy (which allows the routine to find the lowest energy configuration and not merely a local energy minimum). Such a random change is likely to occur when the temperature is high. The probability of this last expression being true (for any value of deltaEnergy) decreases with each iteration of the loop (because `temperature` decreases with every loop). If either of these conditions are true, the junction and edge are added to the seedling path, the energy for the configuration is updated, the length of the seedling path is adjusted, and possibly an angle and/or the separation points, etc. are added:

```
    if deltaEnergy < 0 or exp(-deltaEnergy / (temperatureConst
* temperature)) > random_number_between(0,1)
        mark that this seedling has occupied by this seedling;
        add choice to the primary path of the current seedling;
        if changeAngle is true
            add angle to endAngle;
        end if;
        add the edge length to the seedling path length;
        if newseparation is true
            mark choice has the hypocotyl/radicle separation
point (RHseparation) for the current seedling;
            set separation length of this seedling to the path
length of this seedling;
        end if;
        if separationcomplete is true
            mark separation end point (RHseparation2) as
choice;
        end if;
```

```
            energy = energy + deltaEnergy;
        end if;
```

Whatever happens in each iteration of the simulated annealing loop, the `temperature` is decreased to change the probability of accepting a proposal to change from the current configuration to a configuration having higher energy:

```
        temperature = 0.99 * temperature;
    END
```

The result of the foregoing routine is a list of primary paths, one primary path for each of the seedlings separated, i.e., separately identified, by that routine. Each of the primary paths determined by the routine at task 248 is then processed by software routines 232 and 250 to determine the hypocotyl length, radicle length, and total length.

After the last seedling skeleton in the filtered List of Seedling Skeletons has been processed by the routine of Figure 12 to determine the hypocotyl length, radicle length, and total length, program control returns to task 234 to determine the seedling statistics and the vigor index for these this particular scanned blotter of seedlings. The vigor index is defined as follows:

$$vigor = w_G * growth + w_U * uniformity,$$

$$growth = \min(w_h * \overline{l_h} + w_r * \overline{l_r}, 1000),$$

$$uniformity = \max(1000 - (w_{h'} * s_h + w_{r'} * s_r + w_{total} * s_{total} + w_{r/h} * s_{r/h}) - w_d * numdead, 0)$$

where $\overline{l_h}$ and $\overline{l_r}$ are the sample averages (sample means) of the hypocotyl length and the radicle length, respectively, $s_h$, $s_r$, $s_{total}$, and $s_{r/h}$ are the sample standard deviations of the hypocotyl length, radicle length, total length, and the ratio of the hypocotyl and radicle lengths, and the $w$'s represent associated weights with the parameters being multiplied. The present implementation of the present invention uses the following constants: $w_G = 0.7$, $w_U = 0.3$, $w_h = 2.5$, $w_r = 5.0$, $w_{h'} = 0.75$, $w_{r'} = 0.5$, $w_{total} = 2.5$, and $w_{r/h} = 50$. Other weights will provide satisfactory results and these weights (and other constants used herein) are preferably modifiable by the user to take into account user preference, analysis of different species, etc.

The vigor index is divided into *growth* and *uniformity* parameters because seed analysts examine these to determine seed vigor. Each component has a minimum value of 0 and a maximum value of 1000. The vigor index ranges from 0 to 1000 as it is a weighted average of the components, where weights range from 0 to 1 and sum to 1. Weights can be adjusted to favor either growth or uniformity. Ideally, seedlings should be long and have uniform length. Since uniformity is a component of the vigor index, a uniformly dead sample can have a uniformity of 1000, and thus obtains a relatively high vigor index. This is not desirable. Thus, the software uses a penalty term in uniformity for dead seeds (seeds that were removed from further processing by task 216) so that when a sample has all dead seeds, the uniformity is calculated as 0 instead of 1000.

Next, a graphical overlay is determined from the seedling skeleton paths, at task 236, and displayed along with the vigor index and other parameters at task 238. The graphical overlay is best explained with reference to Figure 9, which shows an image of the original scan of seedlings in the "Seedling Image" window, with red overlaid skeletons indicating where the software of the present invention determined the seedling hypocotyls to be and green skeletons indicating where the software of the present invention determined the seedling radicles to be. The overlay can be one pixel or several pixels in width. The point between the red and green overlays indicates the determined hypocotyl/radicle separation point for each seedling.

Also in the display of Figure 9, there is a region for display of "Individual Measurements" for a particular seedling. If a user places the active point of a pointing device, e.g., places the active point of a cursor of a mouse 20, over one of the images of the one of the seedlings in the Seedling Image window, and actuates a switch of the pointing device, e.g., presses a button on the mouse 20, the software of the present invention displays in the "Individual Measurements" region the hypocotyl length, the radicle length, the total length, and the ratio for that particular seedling.

While the present invention has been illustrated by the description of embodiments thereof, and while the embodiments have been described in considerable detail, it is not the intention of the applicant to restrict or in any way limit the scope of the appended claims to

such detail. Additional advantages and modifications will readily appear to those skilled in the art. For example, the present implementation of the present invention has been optimized for lettuce plants and should function with little or no modification on other plants with similar seedlings, e.g., certain impatiens or soybeans. Seedlings of other plants can be analyzed using the teachings and benefits of the present invention with certain modifications to the code described herein. Therefore, the invention in its broader aspects is not limited to the specific details, representative apparatus and method, and illustrative examples shown and described. Accordingly, departures may be made from such details without departing from the spirit or scope of the applicant's general inventive concept.

We claim:

1. A method of automatically analyzing at least one seedling germinated from at least one seed, comprising the steps of:

(a) capturing a digital image of the at least one seedling;

(b) identifying the at least one seedling in the captured digital image;

(c) determining a primary path of the at least one seedling;

(d) determining at least one value from the primary path of the at least one seedling; and

(e) determining a seed vigor index from at least the at least one value determined from the primary path of the at least one seedling.

2. The method of automatically analyzing at least one seedling according to claim 1:

(a) wherein said step of determining at least one value from the primary path of the at least one seedling comprises the step of determining a value corresponding to an overall length of the at least one seedling from the primary path of the at least one seedling; and

(b) wherein said step of determining a seed vigor index from at least the at least one value determined from the primary path of the at least one seedling comprises the step of determining a seed vigor index from at least the value corresponding to the overall length of the at least one seedling.

3. The method of automatically analyzing at least one seedling according to claim 1 further comprising the step of determining a separation point between the hypocotyl of the at least one seedling and the radicle of the at least one seedling; and:

(a) wherein said step of determining at least one value from the primary path of the at least one seedling comprises the step of determining a value corresponding to the length of at least one of the hypocotyl of the at least one seedling and the radicle of the at least one seedling; and

(b) wherein said step of determining a seed vigor index from at least the at least one value determined from the primary path of the at least one seedling comprises the step of determining a seed vigor index from at least the value corresponding to the length of at least one of the hypocotyl of the at least one seedling and the radicle of the at least one seedling.

4. The method of automatically analyzing at least one seedling according to claim 1 further comprising the step of determining a separation point between the hypocotyl of the at least one seedling and the radicle of the at least one seedling; and:

(a) wherein said step of determining at least one value from the primary path of the at least one seedling comprises the step of determining a hypocotyl length value corresponding to the length of the hypocotyl of the at least one seedling and a radicle length value corresponding to the length of the radicle of the at least one seedling; and

(b) wherein said step of determining a seed vigor index from at least the at least one value determined from the primary path of the at least one seedling comprises the step of determining a seed vigor index from at least the hypocotyl length value and the radicle length value.

5. The method of automatically analyzing at least one seedling according to claim 1 wherein said step of determining a primary path of the at least one seedling comprises the step of determining a locus of pixels, the locus of pixels corresponding to the primary path of the at least one seedling and the locus of pixels being narrower in width than the width of the at least one seedling in the digital image of the at least one seedling.

6. The method of automatically analyzing at least one seedling according to claim 1 wherein said step of determining a primary path of the at least one seedling comprises the step of determining a locus of pixels, the locus of pixels corresponding to the primary path of the at least one seedling and the locus of pixels being a predetermined number of pixels in width.

7. The method of automatically analyzing at least one seedling according to claim 1 wherein said step of determining a primary path of the at least one seedling comprises the step of determining a locus of pixels, the locus of pixels corresponding to the primary path of the at least one seedling and the locus of pixels being one pixel in width.

8. The method of automatically analyzing at least one seedling according to claim 1 further comprising the step of separately identifying a plurality of overlapped seedlings in the digital image of the at least one seedling, and

(a) wherein said step of determining a primary path of the at least one seedling comprises the step of determining a primary path for each of the separately identified overlapped seedlings;

(b) wherein said step of determining at least one value from the primary path of the at least one seedling comprises the step of determining from the primary path for each of the separately identified overlapped seedlings a value corresponding to an overall length of that separately identified overlapped seedling; and

(c) wherein said step of determining a seed vigor index from at least the at least one value determined from the primary path of the at least one seedling comprises the step of determining a seed vigor index from at least the plurality of values determined in step (b).

9. The method of automatically analyzing at least one seedling according to claim 1 further comprising the step of separately identifying a plurality of overlapped seedlings in the digital image of the at least one seedling, and

(a) wherein said step of determining a primary path of the at least one seedling comprises the step of determining a primary path for each of the separately identified overlapped seedlings;

(b) wherein said step of determining at least one value from the primary path of the at least one seedling comprises the step of determining from the primary path for each of the separately identified overlapped seedlings a value corresponding to the length of at least one of the hypocotyl of that separately identified overlapped seedling and the radicle of that separately identified overlapped seedling; and

(c) wherein said step of determining a seed vigor index from at least the at least one value determined from the primary path of the at least one seedling comprises the step of determining a seed vigor index from at least the plurality of values determined in step (b).

10. The method of automatically analyzing at least one seedling according to claim 1 further comprising the step of separately identifying a plurality of overlapped seedlings in the digital image of the at least one seedling, and

(a) wherein said step of determining a primary path of the at least one seedling comprises the step of determining a primary path for each of the separately identified overlapped seedlings;

(b) wherein said step of determining at least one value from the primary path of the at least one seedling comprises the step of determining from the primary path for each of the separately identified overlapped seedlings a hypocotyl length value corresponding to the length of the hypocotyl of that separately identified overlapped seedling and a radicle length value corresponding to the length of the radicle of that separately identified overlapped seedling; and

(c) wherein said step of determining a seed vigor index from at least the at least one value determined from the primary path of the at least one seedling comprises the step of determining a seed vigor index from at least the plurality of hypocotyl length values determined in step (b) and at least the plurality of radicle length values determined in step (b).

11. The method of automatically analyzing at least one seedling according to claim 8 wherein said step of separately identifying a plurality of overlapped seedlings in the digital image of the at least one seedling comprises evaluating an energy function.

12. The method of automatically analyzing at least one seedling according to claim 9 wherein said step of separately identifying a plurality of overlapped seedlings in the digital image of the at least one seedling comprises evaluating an energy function.

13. The method of automatically analyzing at least one seedling according to claim 10 wherein said step of separately identifying a plurality of overlapped seedlings in the digital image of the at least one seedling comprises evaluating an energy function.

14. The method of automatically analyzing at least one seedling according to claim 8 wherein said step of separately identifying a plurality of overlapped seedlings in the digital image of the at least one seedling comprises evaluating an energy function based on proposed configurations of at least partial primary paths of overlapped seedlings.

15. The method of automatically analyzing at least one seedling according to claim 9 wherein said step of separately identifying a plurality of overlapped seedlings in the digital image of the at least one seedling comprises evaluating an energy function based on proposed configurations of at least partial primary paths of overlapped seedlings.

16. The method of automatically analyzing at least one seedling according to claim 10 wherein said step of separately identifying a plurality of overlapped seedlings in the digital image of the at least one seedling comprises evaluating an energy function based on proposed configurations of at least partial primary paths of overlapped seedlings.

17. The method of automatically analyzing at least one seedling according to claim 8 wherein said step of separately identifying a plurality of overlapped seedlings in the digital

image of the at least one seedling comprises evaluating an energy function based on proposed configurations of at least partial primary paths of overlapped seedlings using the following heuristics: primary paths do not make unnaturally sharp turns and seedling edges should be used as much as possible.

18. The method of automatically analyzing at least one seedling according to claim 9 wherein said step of separately identifying a plurality of overlapped seedlings in the digital image of the at least one seedling comprises evaluating an energy function based on proposed configurations of at least partial primary paths of overlapped seedlings using the following heuristics: primary paths do not make unnaturally sharp turns and seedling edges should be used as much as possible.

19. The method of automatically analyzing at least one seedling according to claim 10 wherein said step of separately identifying a plurality of overlapped seedlings in the digital image of the at least one seedling comprises evaluating an energy function based on proposed configurations of at least partial primary paths of overlapped seedlings using the following heuristics: primary paths do not make unnaturally sharp turns and seedling edges should be used as much as possible.

20. The method of automatically analyzing at least one seedling according to claim 8 wherein said step of separately identifying a plurality of overlapped seedlings in the digital image of the at least one seedling comprises evaluating an energy function based on proposed configurations of at least partial primary paths of overlapped seedlings using the following heuristics: primary paths should not make unnaturally sharp turns, seedling edges should be used as much as possible, and all primary axes should have a hypocotyl/radicle separation point.

21. The method of automatically analyzing at least one seedling according to claim 9 wherein said step of separately identifying a plurality of overlapped seedlings in the digital image of the at least one seedling comprises evaluating an energy function based on proposed configurations of at least partial primary paths of overlapped seedlings using the following heuristics: primary paths should not make unnaturally sharp turns, seedling edges should be used as much as possible, and all primary axes should have a hypocotyl/radicle separation point.

22. The method of automatically analyzing at least one seedling according to claim 10 wherein said step of separately identifying a plurality of overlapped seedlings in the digital image of the at least one seedling comprises evaluating an energy function based on proposed configurations of at least partial primary paths of overlapped seedlings using the following heuristics: primary paths should not make unnaturally sharp turns, seedling edges should be used as much as possible, and all primary axes should have a hypocotyl/radicle separation point.

23. The method of automatically analyzing at least one seedling according to claim 1 further comprising the steps of determining a first locus of points indicating the hypocotyl of the at least one seedling, determining a second locus of points indicating the radicle of the at least one seedling, overlaying the first and second loci over an image of the seedlings to generate a composite image, and displaying the composite image.

24. The method of automatically analyzing at least one seedling according to claim 23 wherein said step of displaying the composite image comprises the step of displaying the composite image on a video display terminal.

25. The method of automatically analyzing at least one seedling according to claim 23 wherein said step of displaying the composite image comprises the step of printing the image on a printer or plotter.

26. A method of automatically analyzing at least one seedling germinated from at least one seed, comprising the steps of:

(a) capturing a digital image of the at least one seedling;

(b) determining a first locus of points indicating the hypocotyl of the at least one seedling;

(c) determining a second locus of points indicating the radicle of the at least one seedling;

(d) overlaying the first and second loci over an image of the seedlings to generate a composite image; and

(e) displaying the composite image.

27. A method of analyzing at least one seedling germinated from at least one seed, comprising the steps of:

(a) placing a growing medium in a shallow container;

(b) wetting the growing medium;

(c) placing the at least one seed onto the growing medium;

(d) germinating the at least one seed with the shallow container at an angle with respect to the vertical that is less than about 10°;

(e) capturing a digital image of the at least one seedling; and

(f) analyzing the captured digital image of the germinated seedling.

28. The method of analyzing at least one seedling according to claim 1 wherein said step of germinating the at least one seed with the shallow container at an angle with respect to the

vertical that is less than about 10° comprises the step of positioning the shallow container vertically.

29. The method of analyzing at least one seedling according to claim 1 wherein said step of capturing a digital image of the at least one seedling comprises capturing an image of the at least one seedling using a scanner having a scanner surface and positioned with its scanner surface oriented at least 90° from the horizontal.

30. The method of analyzing at least one seedling according to claim 1 wherein said step of capturing a digital image of the at least one seedling comprises capturing an image of the at least one seedling using a scanner having a scanner surface and positioned with its scanner surface substantially inverted so that it captures an image of at least one seedling positioned beneath the scanner.

ABSTRACT

A system and method for automatically determining a seed vigor index for a lot of seeds by analysis of a scanned image of a plurality of seedlings grown from lot of seeds, including automatically separating and analyzing overlapped seedlings. According to one aspect of the current invention, seedling analysis software is used to analyze an image of seedlings. The seedling analysis software preferably analyzes both hypocotyl and radicle lengths and thus determines the separation point between the two for each seedling. The seedling analysis software also preferably separates overlapped seedlings, preferably using a simulated annealing technique. According to another aspect of the present invention, a low-cost scanner placed in an inverted configuration in a scanner enclosure is used to generate high-quality, reproducible images of seedlings. According to yet another aspect of the present invention, a method of using ordinary germination boxes, i.e., "sandwich boxes" is used to germinate seedlings that greatly facilitate computer-based analysis. In general, this method comprises placing germination blotter paper in the lid of a sandwich box and growing seedlings within the sandwich box in a nearly vertical (upright) position in a darkened germination chamber. The resulting seedlings produce hypocotyls that grow essentially upward and radicles that grow essentially downward, which greatly facilitates image acquisition of the entire seedling, e.g., with an inverted scanner.

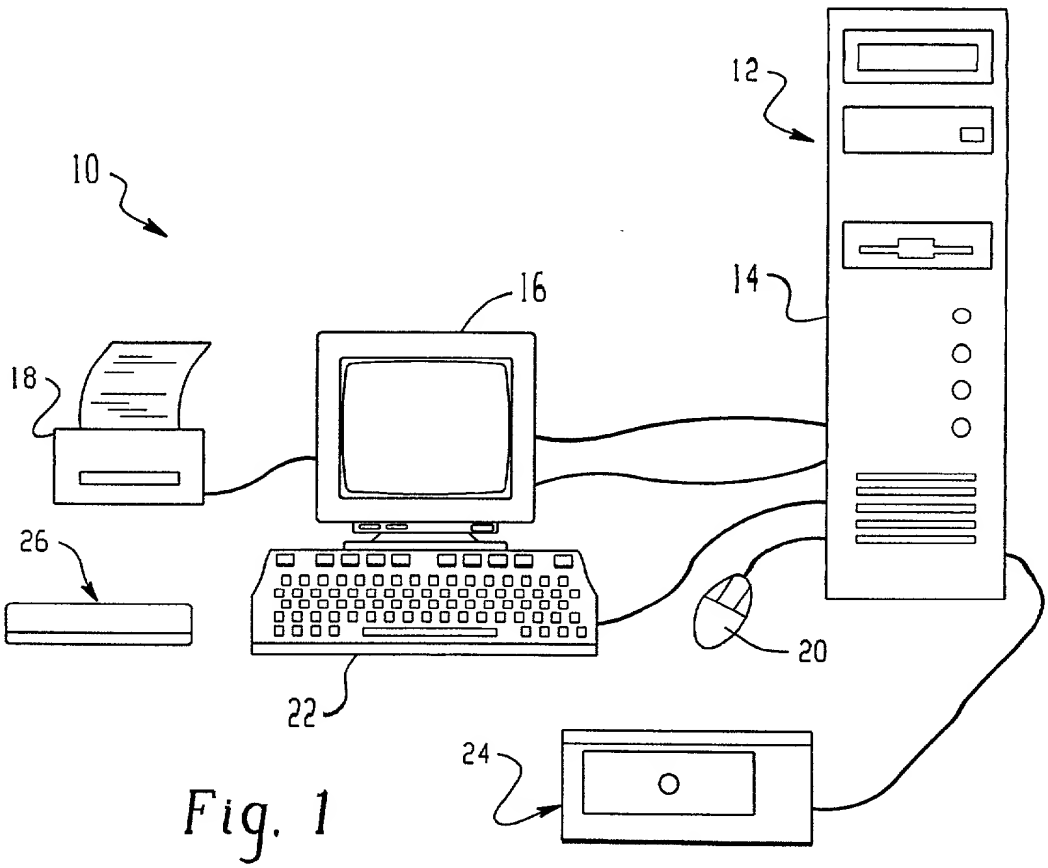


Fig. 1

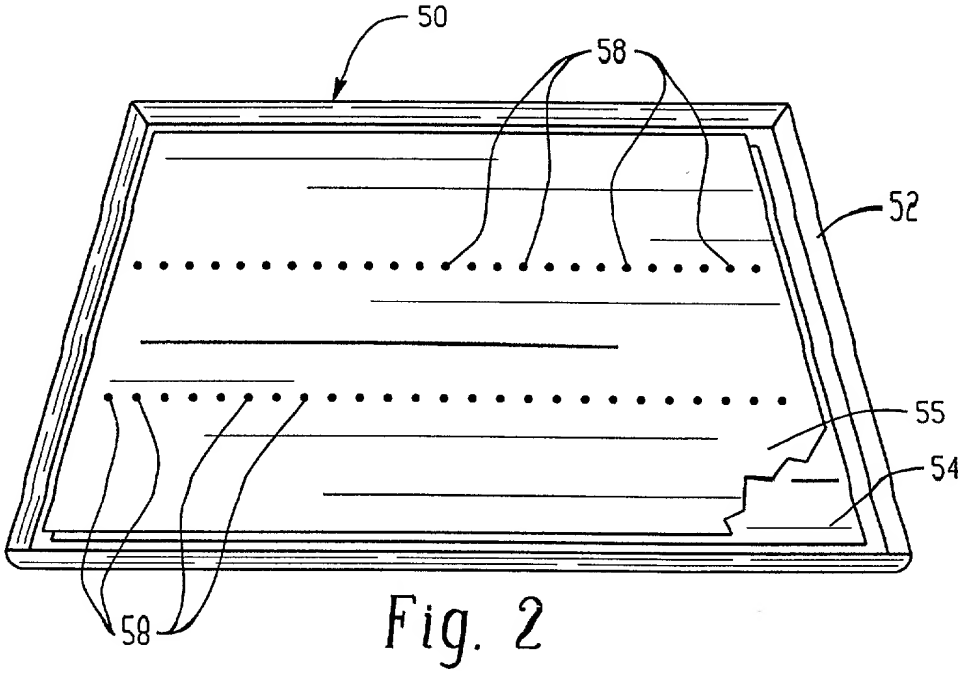


Fig. 2

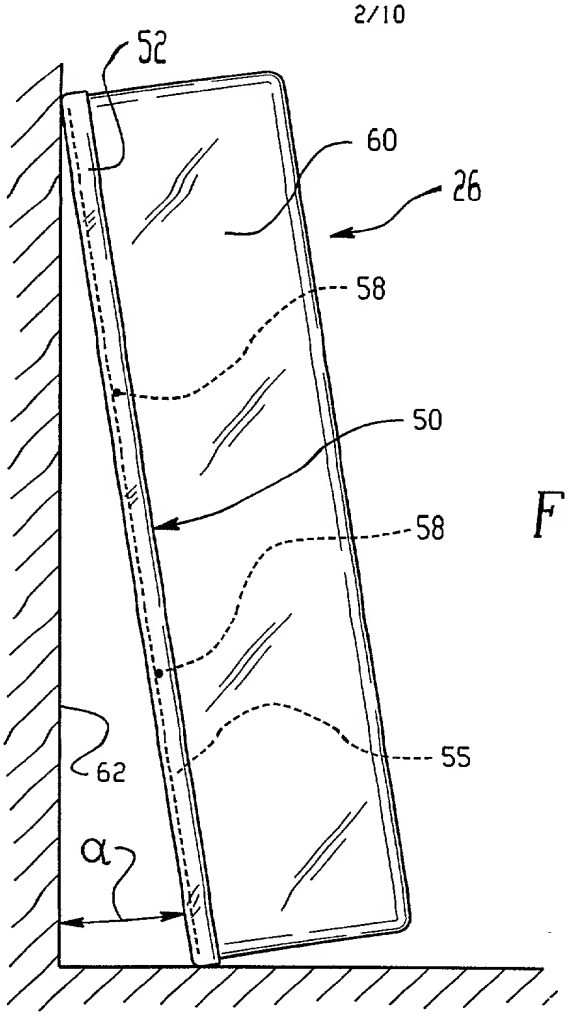
52
60
26
58
50
58
55
62
α

Fig. 3

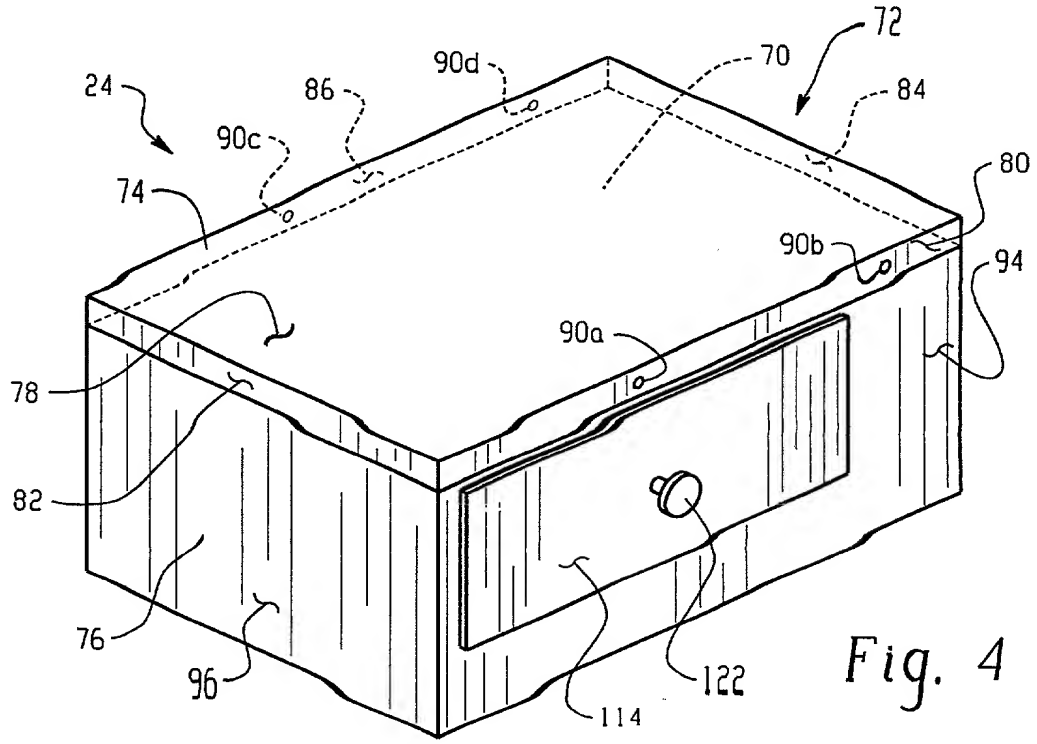
72
70
90d
24
86
84
90c
80
74
90b
94
90a
78
82
76
96
122
114

Fig. 4

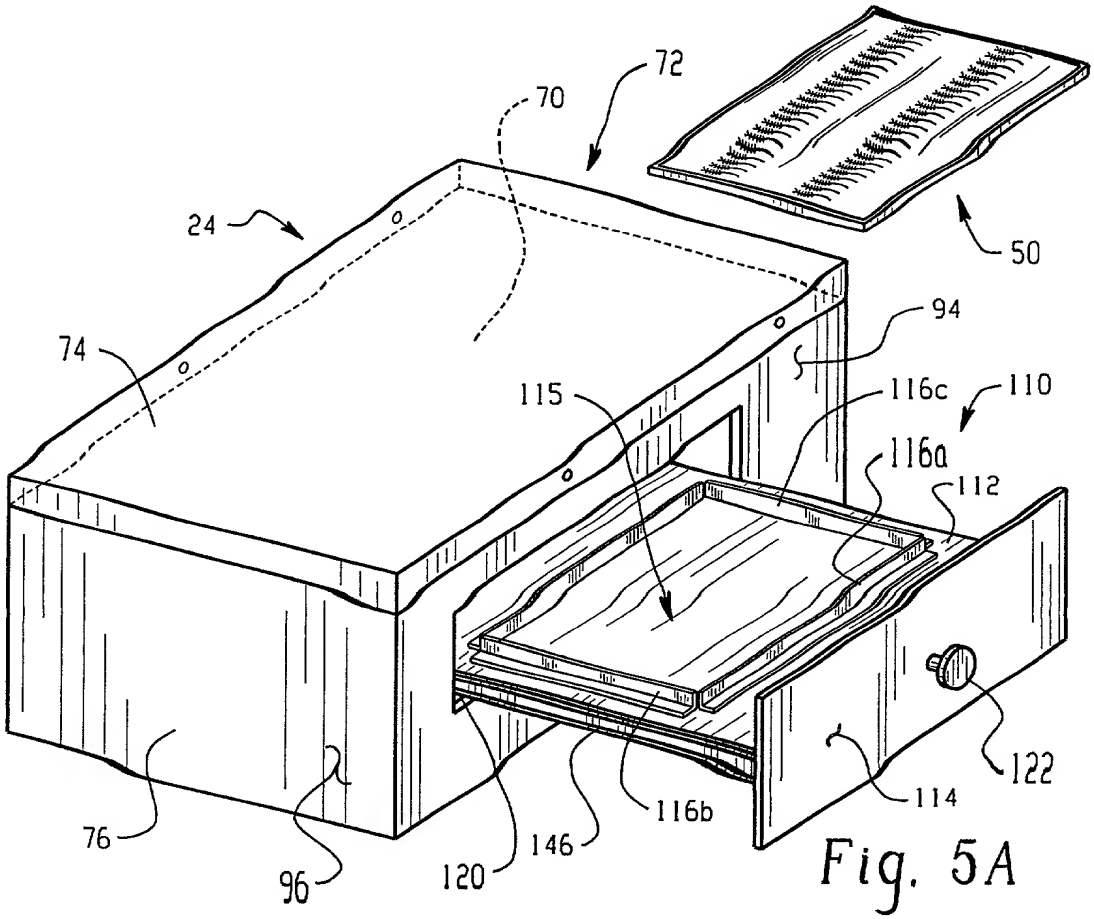
72
70
24
50
74
94
115
116c
110
116a
112
122
114
116b
76
96
120
146

Fig. 5A

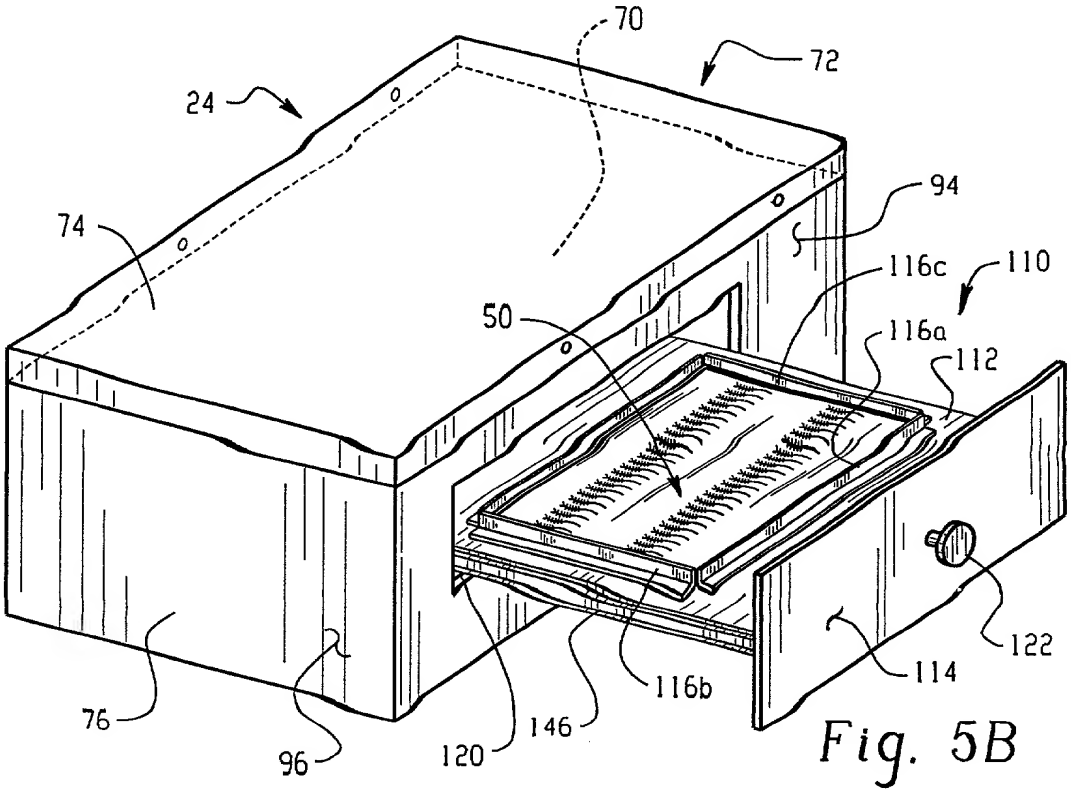
70
72
24
74
94
116c
110
50
116a
112
122
114
116b
76
96
120
146

Fig. 5B

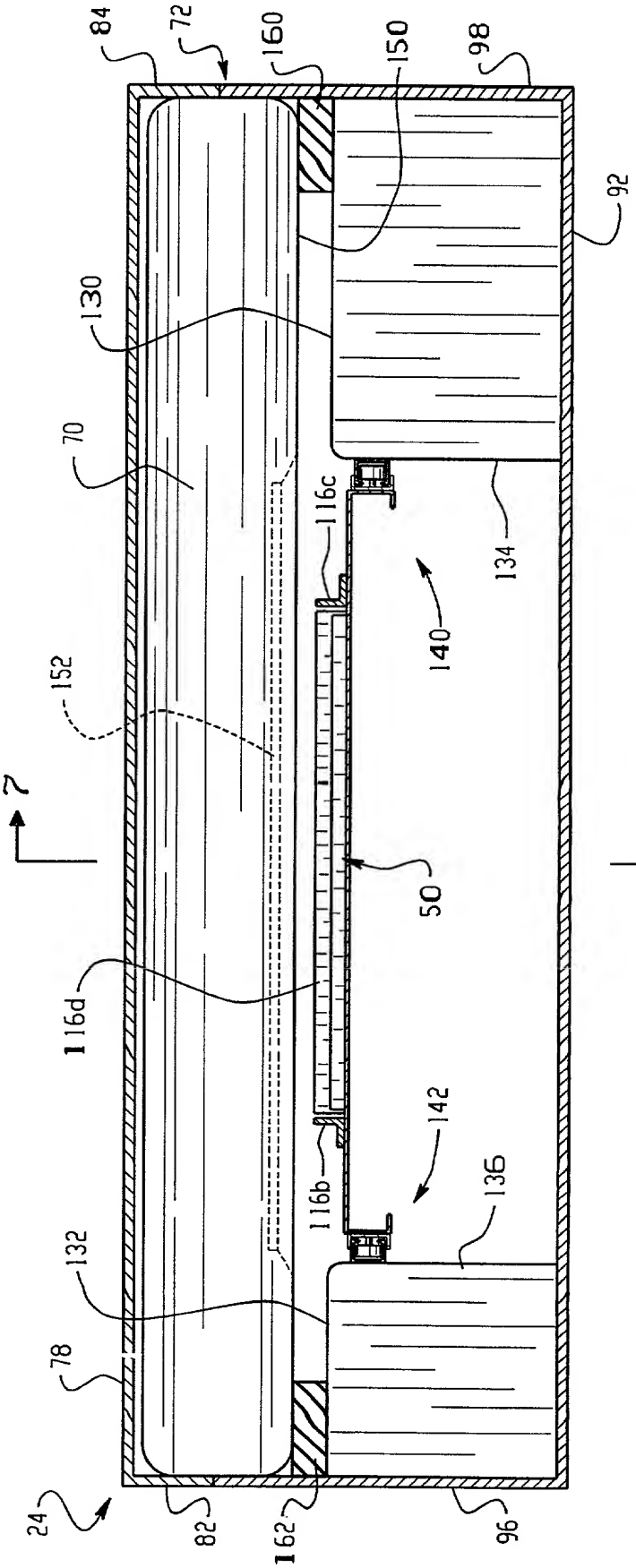
84
72
160
150
98
92
130
70
116c
134
140
152
7
7
50
116d
142
116b
136
132
78
24
82
162
96

Fig. 6

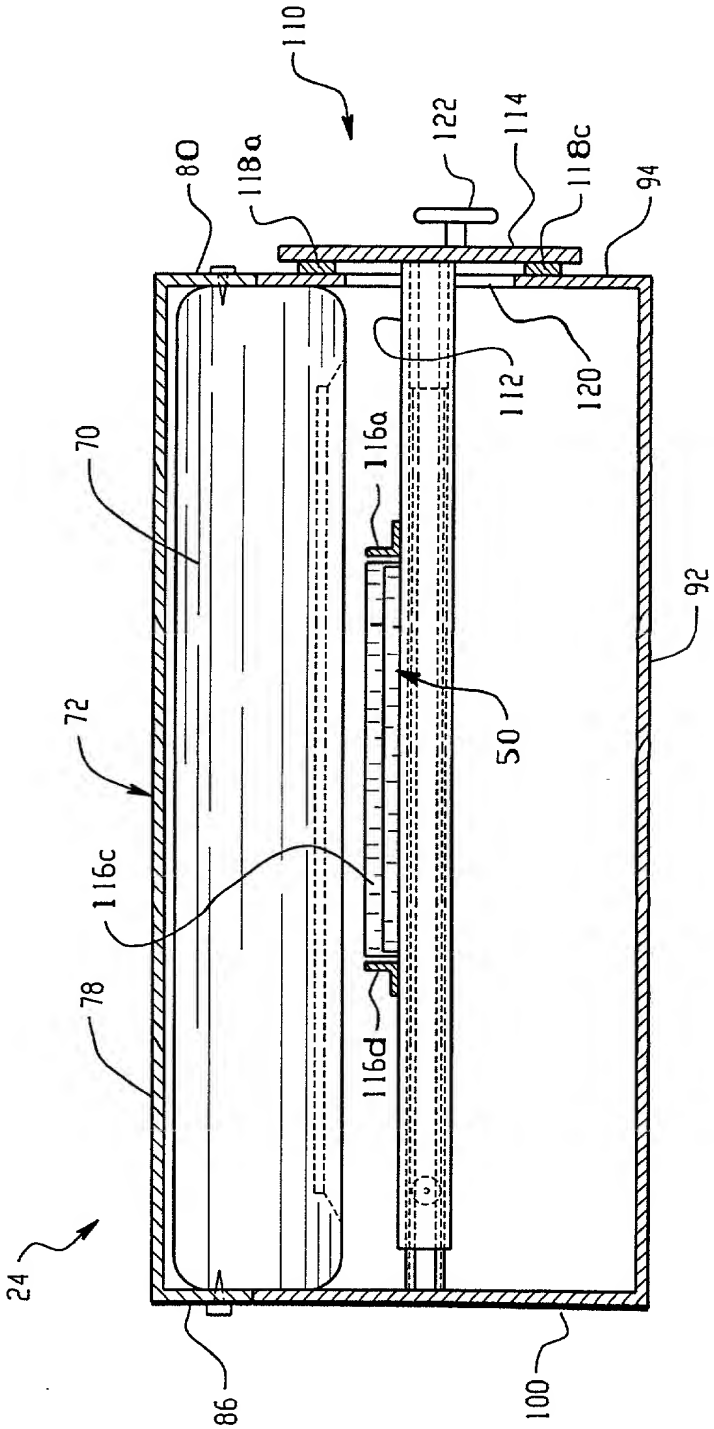
110
80
118a
122
114
118c
94
112
120
116a
70
92
72
50
116c
78
116d
24
86
100

Fig. 7A

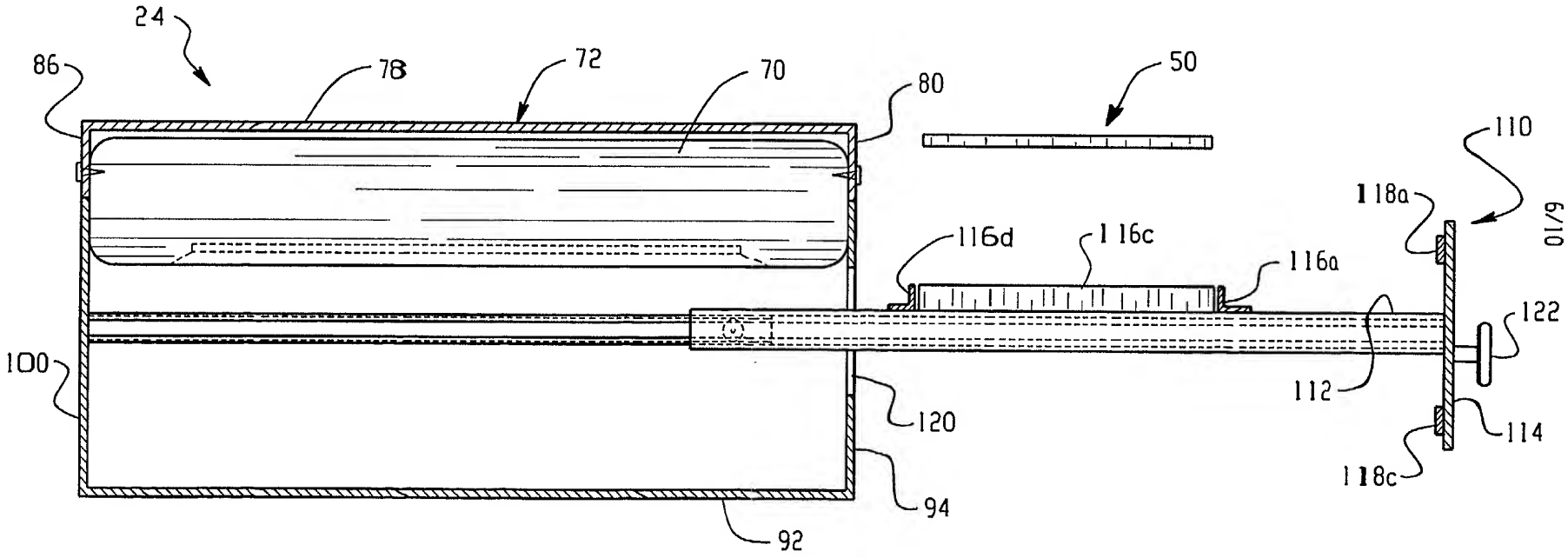
24
86
78
72
70
80
50
110
118a
116d
116c
116a
122
100
112
120
114
118c
94
92

Fig. 7B

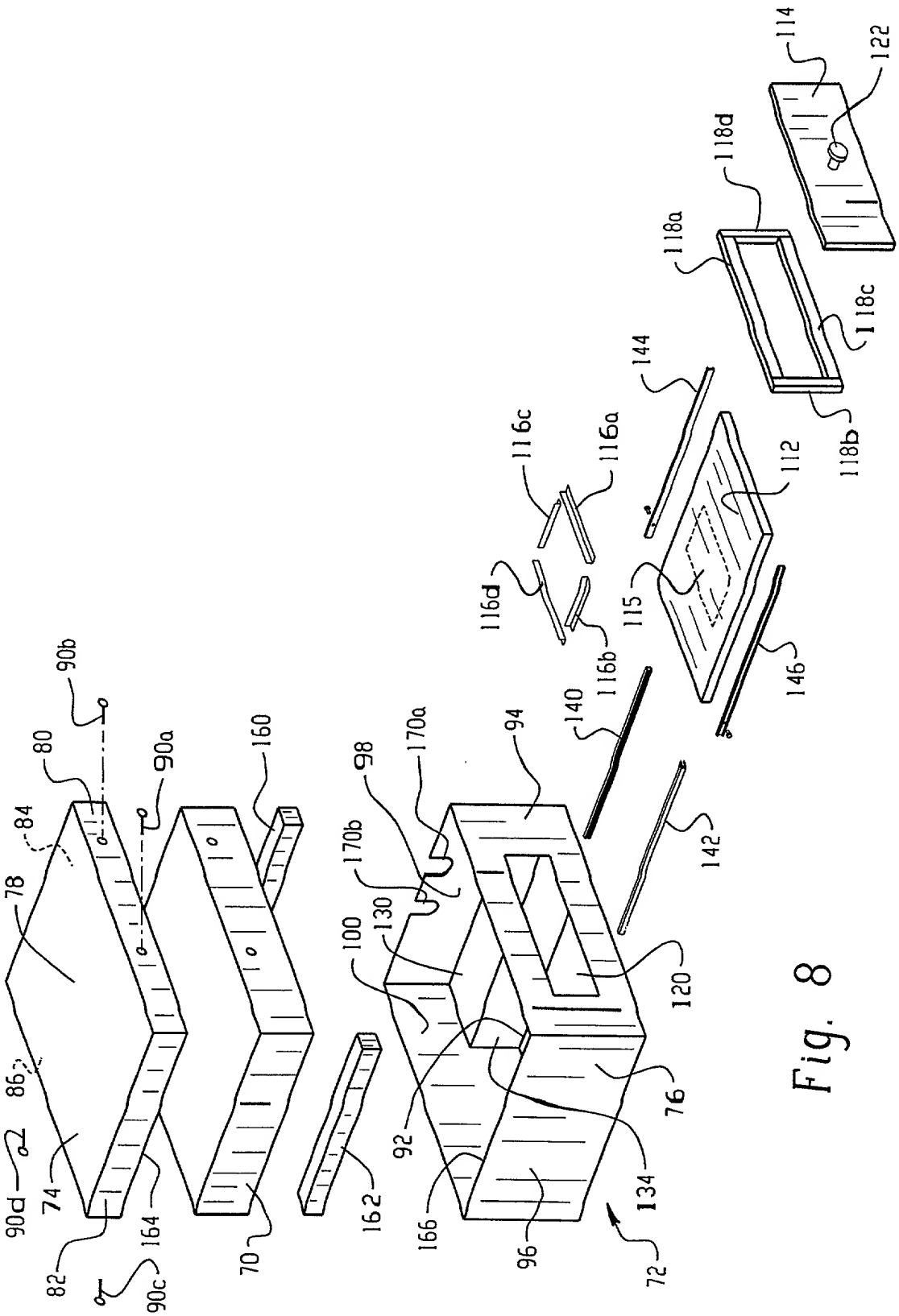
90b
80
90a
84
78
160
86
90d
74
164
82
90c
70
162
98
170a
94
170b
100
130
92
166
96
72
134
76
120
140
142
116b
116d
116c
116a
144
115
112
146
118a
118d
118c
118b
114
122

Fig. 8

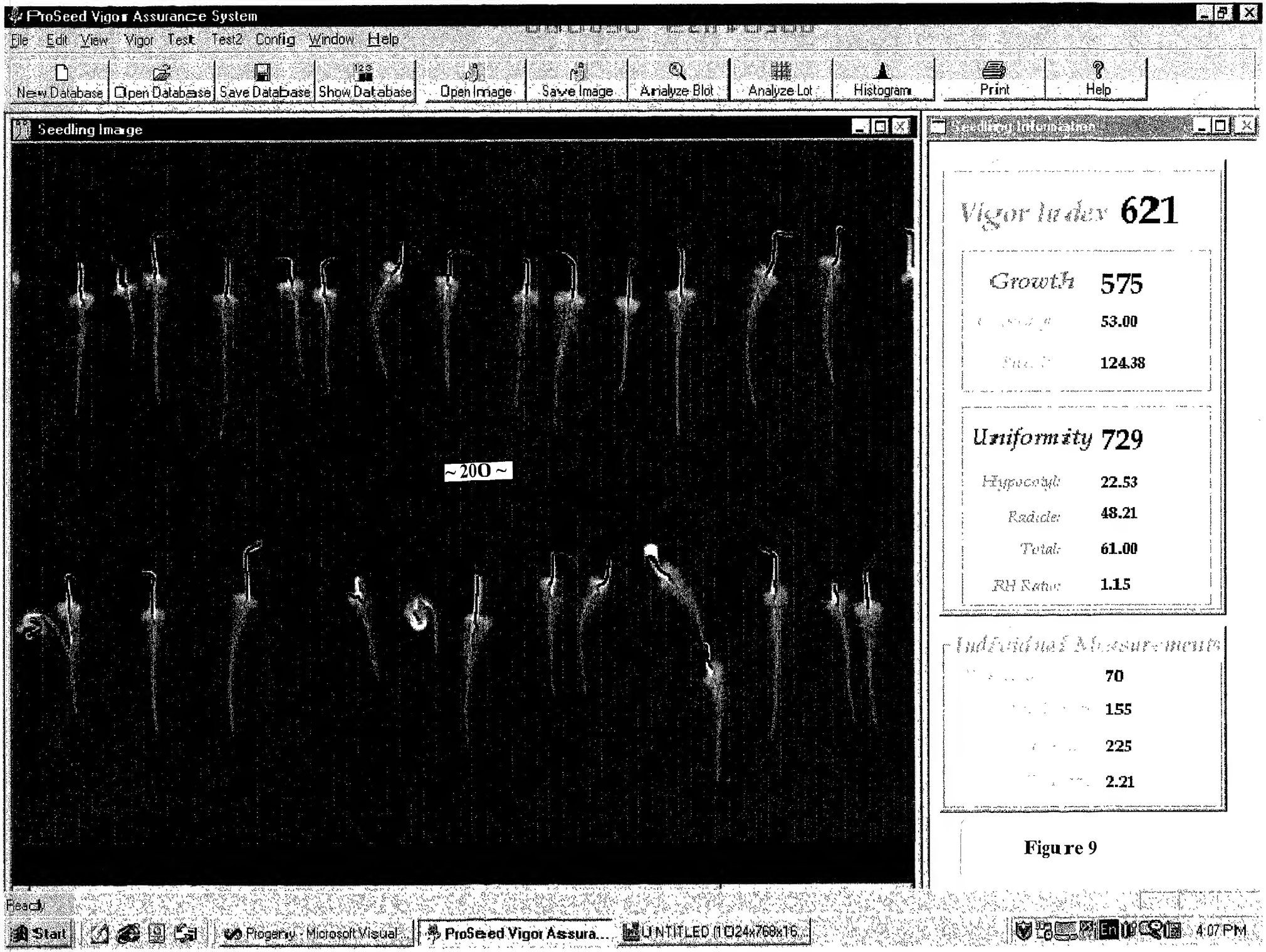
ProSeed Vigor Assurance System
File Edit View Vigor Test Test2 Config Window Help
New Database
Open Database
Save Database
Show Database
Open Image
Save Image
Analyze Blot
Analyze Lot
Histogram
Print
Help
Seedling Image
~200~
Seedling Information
Vigor Index 621
Growth 575
53.00
124.38
Uniformity 729
Hypocotyl 22.53
Radicle 48.21
Total 61.00
RH Ratio 1.15
Individual Measurements
70
155
225
2.21
Ready
Start
Progeny - Microsoft Visual...
ProSeed Vigor Assura...
UNTITLED (1024x768x16...
4:07 PM

Figure 9

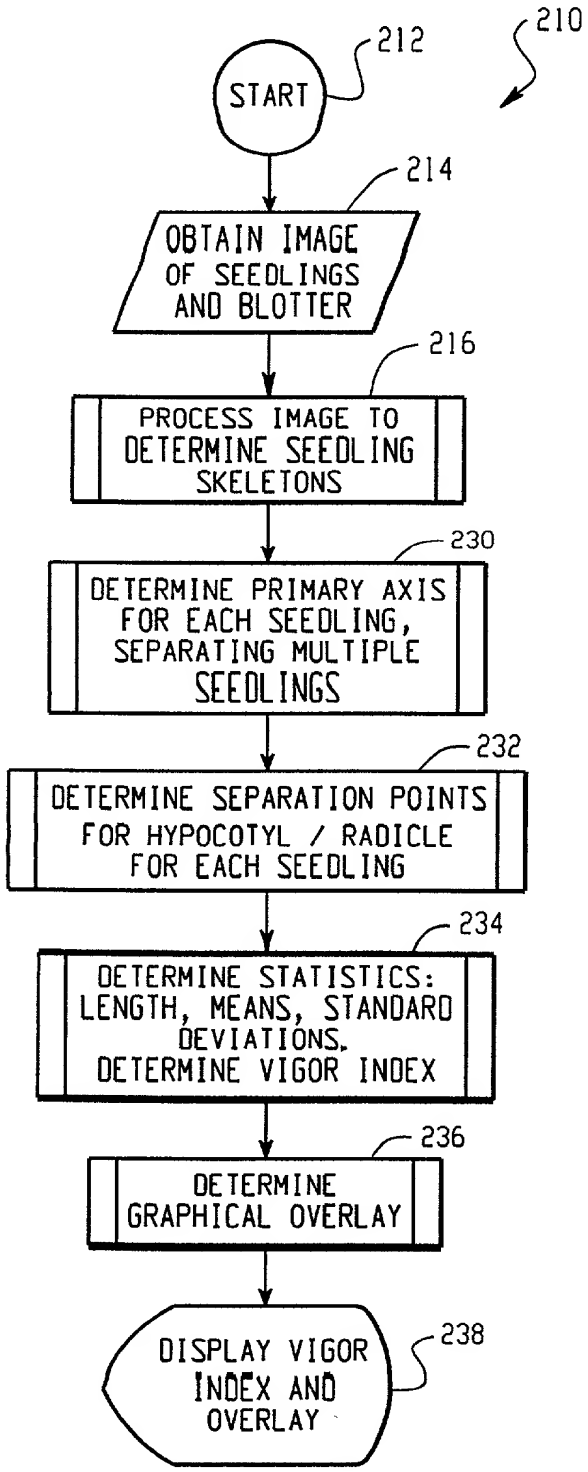
210
212
START
214
OBTAIN IMAGE OF SEEDLINGS AND BLOTTER
216
PROCESS IMAGE TO DETERMINE SEEDLING SKELETONS
230
DETERMINE PRIMARY AXIS FOR EACH SEEDLING, SEPARATING MULTIPLE SEEDLINGS
232
DETERMINE SEPARATION POINTS FOR HYPOCOTYL / RADICLE FOR EACH SEEDLING
234
DETERMINE STATISTICS: LENGTH, MEANS, STANDARD DEVIATIONS. DETERMINE VIGOR INDEX
236
DETERMINE GRAPHICAL OVERLAY
238
DISPLAY VIGOR INDEX AND OVERLAY

Fig. 10

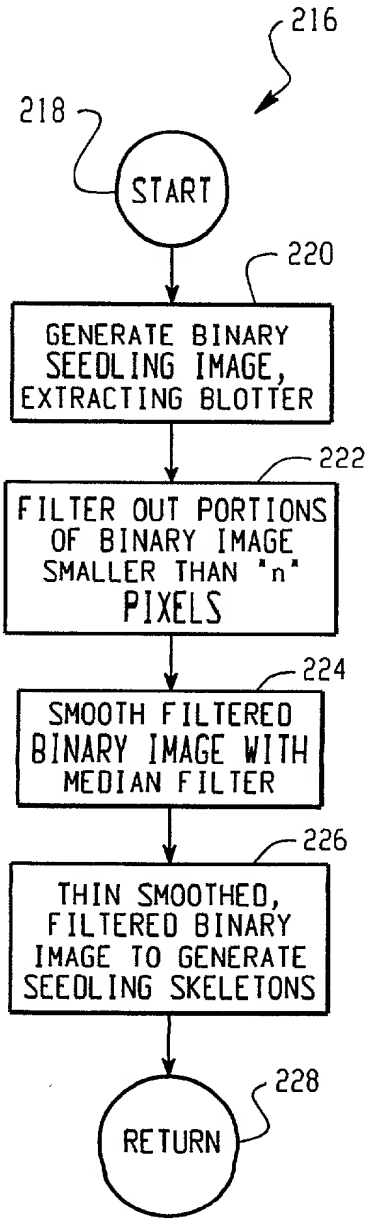
216
218
START
220
GENERATE BINARY SEEDLING IMAGE, EXTRACTING BLOTTER
222
FILTER OUT PORTIONS OF BINARY IMAGE SMALLER THAN "n" PIXELS
224
SMOOTH FILTERED BINARY IMAGE WITH MEDIAN FILTER
226
THIN SMOOTHED, FILTERED BINARY IMAGE TO GENERATE SEEDLING SKELETONS
228
RETURN

Fig. 11

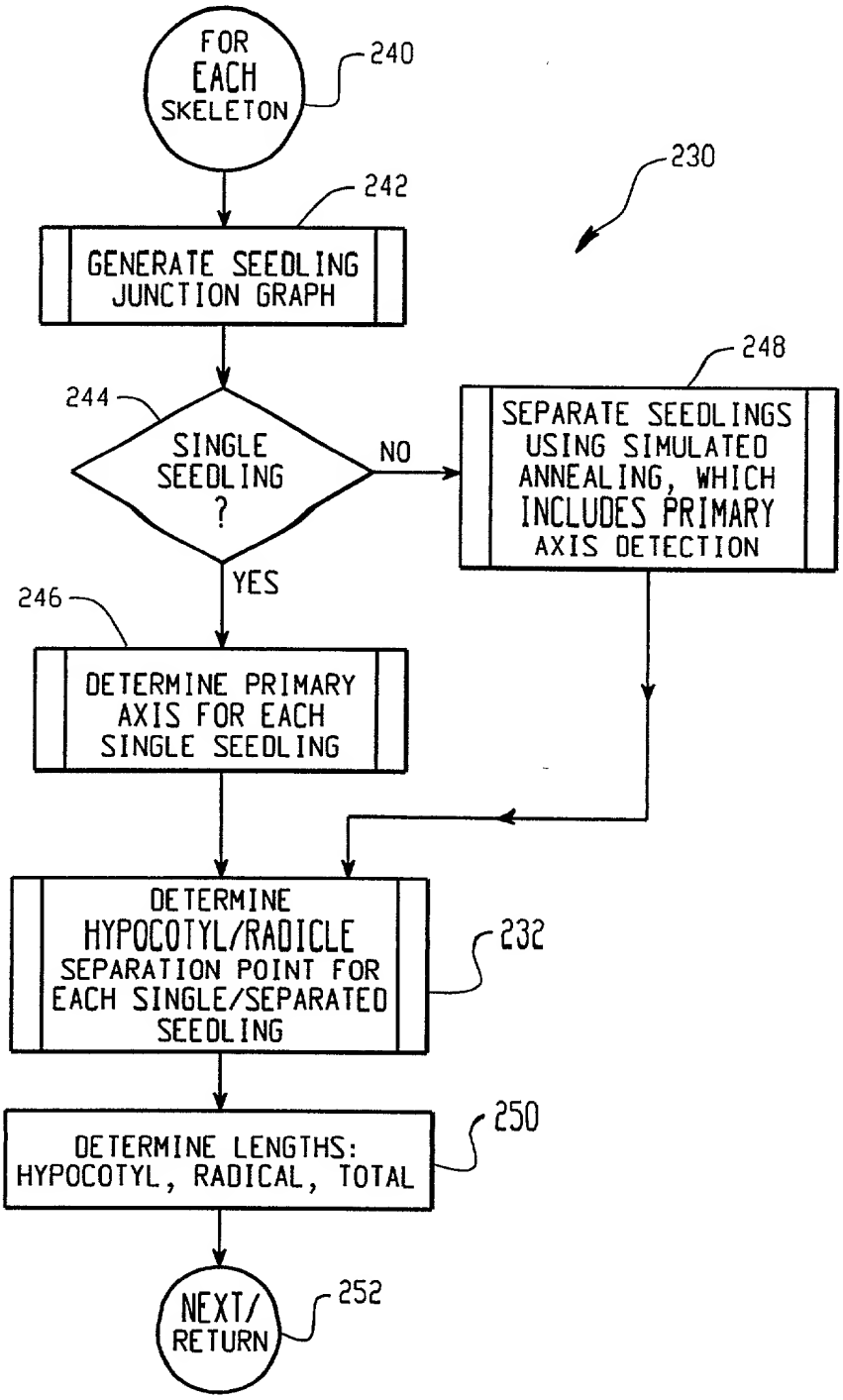
FOR EACH SKELETON
240
230
242
GENERATE SEEDLING JUNCTION GRAPH
248
244
SINGLE SEEDLING ?
NO
SEPARATE SEEDLINGS USING SIMULATED ANNEALING, WHICH INCLUDES PRIMARY AXIS DETECTION
YES
246
DETERMINE PRIMARY AXIS FOR EACH SINGLE SEEDLING
DETERMINE HYPOCOTYL/RADICLE SEPARATION POINT FOR EACH SINGLE/SEPARATED SEEDLING
232
DETERMINE LENGTHS: HYPOCOTYL, RADICAL, TOTAL
250
NEXT/ RETURN
252

Fig. 12

CALFEE, HALTER & GRISWOLD LLP
Docket No. 22727/04060

# DECLARATION
# AND POWER OF ATTORNEY
## ORIGINAL APPLICATION

As below named inventors, we hereby declare that:

Our residences, post office addresses and citizenships are as stated below next to our names.

We believe we are the original, first and joint inventors of the subject matter which is claimed and for which a patent is sought on the invention entitled:

**SYSTEM AND METHOD FOR AUTOMATICALLY DETERMINING SEED VIGOR INDEX FROM SEEDLINGS WITH AUTOMATIC SEPARATION OF OVERLAPPED SEEDLINGS**

the specification of which

☒ is attached hereto,
☐ was filed on _____ as Application Serial No. _____.
☐ and was amended on _____.
(if applicable)

We hereby state that we have reviewed and understand the contents of the above-identified specification, including the claims, as amended by any amendment referred to above.

We acknowledge the duty to disclose information which is material to the examination of this application in accordance with Title 37, code of Federal Regulations, §1.56(a).

We hereby appoint the following attorney(s) to prosecute this application and to transact all business in the Patent and Trademark Office connected therewith:

Charles B. Lyon
Reg. No. 25,739

John T. Wiedemann
Reg. No. 28,920

Jeanne E. Longmuir
Reg. No. 33,133

Tara A. Kastelic
Reg. No 35,980

Mary E. Golrick
Reg. No. 34,829

John S. Cipolla
Reg. No. 37,597

Sean T. Moorhead
Reg. No. 38,564

James A. Rich
Reg. No. 25,519

Nenad Pejic
Reg. No. 37,415

Pamela A. Docherty
Reg. No. 40,591

June E. Rickey
Reg. No. 40,144

Eileen T. Mathews
Reg. No. 41,973

John E. Miller
Reg. No. 26,206

Leonard L. Lewis
Reg. No. 31,176

Ronald D. Gutt
Reg. No. 43,650

Peter Kraguljac
Reg. No. 38,520

S. Paige Christopher
Reg. No. 39,503

Brian D. Johnson
Reg. No. 40, 665

Larry W. Conner
Reg. No. 44,627

William E. Zitelli
Reg. No. 28,551

George R. Hoskins
Reg. No. P46,780

Address all telephone calls to Sean T. Moorhead at telephone number (216) 622-8844.

Address all correspondence to Sean T. Moorhead, CALFEE, HALTER & GRISWOLD, LLP, 1400 McDonald Investment Center, 800 Superior Avenue, Cleveland, Ohio 44114.

We hereby declare that all statements made hereon of our own knowledge are true and that all statements made on information and belief are believed to be true; and further that these statements were made with the knowledge that willful false statements and the like so made are punishable by fine or imprisonment, or both, under §1001 of Title 18 of the United States Code and that such willful false statements may jeopardize the validity of the application or any patent issued thereon.

Full name of first inventor: Miller Baird McDonald, Jr.

Inventor's signature _____ _____
(Date)

Residence: 5837 Tarton Circle
Dublin, Ohio 43017

Country of Citizenship: US

Post Office Address: Same as above

Full name of second inventor: Kikuo Fujimora

Inventor's signature _____ _____
(Date)

Residence: 3077 Middlefield Road #203
Palo Alto, California 94306

Country of Citizenship: Japan

Post Office Address: Hamilton Station Post Office
380 Hamilton Avenue
Palo Alto, California 94306

Full name of third inventor: Mark Alan Bennett

Inventor's signature: _____ _____
(Date)

Residence: 5815 Lou Street
Columbus, Ohio 43231

Country of Citizenship: US

Post Office Address: Same as above

Full name of fourth inventor: Yusaku Sako

Inventor's signature _____ _____
(Date)

Residence: 2076 Wendys Drive, #60
Columbus, Ohio 43220

Country of Citizenship: Japan

Post Office Address: Same as above

Full name of fifth inventor: Andrew Frederick Evans

Inventor's signature _____ _____
(Date)

Residence: 1015 Tipton Court
Westerville, Ohio 43081

Country of Citizenship: US

Post Office Address: Same as above

```
// SeedlingAnalyzer.cpp: implementation of the SeedlingAnalyzer class.
//
//////////////////////////////////////////////////////////////////////

#include "stdafx.h"
#include "Progeny.h"
#include "SeedlingAnalyzer.h"
#include "ProgenyDoc.h"
#include "ColorConversion.h"
#include "const.h"
#include <math.h>
#include <fstream.h>

#ifdef _DEBUG
#undef THIS_FILE
static char THIS_FILE[]=__FILE__;
#define new DEBUG_NEW
#endif

#define GETRAND (rand()/(double)(RAND_MAX+1))

//////////////////////////////////////////////////////////////////////
// Construction/Destruction
//////////////////////////////////////////////////////////////////////

SeedlingAnalyzer::~SeedlingAnalyzer()
{

}

// Returns information needed to measure radicle and hypocotyl lengths on
// each seedling detected from a color image of seedlings.
std::vector<SeedlingInfo *> SeedlingAnalyzer::ProcessSeedlingImage(ImageIO* image)
{
        // Minimum pixel area for a blob to be qualified as a seedling blob
        const int c_minblobsize = 100;
        // kernel size used for median filtering
        const int c_medkernelsize = 3;
        // Minimum skeleton length to be qualified as a seedling blob
        const int c_minseglen = 15;
        // Minimum size for a blob to be considered as a cotyledon
        const int c_mincotsize = 10;
        // Minimum size for a blob to be considered as a seed coat
        const int c_mincoatsize = 10;

        // Seedling separation parameters
        const float c_angleW = 200.0f;
        const float c_initialTemperature = 2000.0f;
        const float c_lengthW = 0.0f;
        const int c_maxIteration = 50000;
        const float c_temperatureConst = 1.0f;
        const float c_unusedW = 10.0f;
        const float c_minEdgeLength = 10.0f;
        const float c_separationW = 200.0f;

        // Data structure to hold extract measurements of seedlings
        std::vector<SeedlingInfo *> seedlinginfo;
```

```
// origimg is the original RGB image of seedlings.
ExImage* origimg = (ExImage*)image;

// Create binary image of cotyledons (leaves)
ExImage* cotimg = origimg->Threshold(200, 255, 200, 255, 0, 200);

// Create binary image of seed coats
ExImage* coatimg = origimg->Threshold(60, 255, 0, 255, 0, 100);

// Binary threshold based on red channel
// To threshold, find the peak intensity on the red channel by FindPeakValue(0).
// The peak intensity + 40 is a good lower bound for thresholding.
ExImage* binimg = origimg->Threshold(origimg->FindPeakValue(0)+40, 255, 0, 255, 0,
255);

// Remove blobs of size less than c_minblobsize
BinaryObjects* binobjs = binimg->GetObjects();
delete binimg;
binobjs->FilterOut(c_minblobsize);
ExImage* cleanbinimg = binobjs->CreateImage();
delete binobjs;

// Perform median filtering to remove noise and smooth out edges

ExImage* smoothimg = cleanbinimg->MedianFilter(c_medkernelsize);
BinaryObjects* smoothobjs = smoothimg->GetObjects();

std::vector<ObjectInfo*> blobobjinfo = smoothobjs->GetObjectInfo();

// 4. Perform thinning (skeletonization)

ExImage* thinimg = smoothimg->Thin();
delete smoothimg;

// Throw out skeletons of length less than c_minsegln
BinaryObjects* binskel = thinimg->GetObjects();
delete thinimg;
binskel->FilterOut(c_minseglen);

// objinfo holds information for all detected objects in the
// skeletonized image.
std::vector<ObjectInfo*> objinfo = binskel->GetObjectInfo();

// Since the extracted skeleton contains root hairs and other
// noise, use the shortest path algorithm to find the primary
// axis.  Also, the second junction in the path is the hypocotyl
// separation, if it is not the end junction (i.e., not the bottom).

ShortestPathFinder spf;
ConnectivityGraph<Edge *>* jg;  // Junction graph.

bool added = false;  // True if seedlinginfo was updated.

// Perform the following loop for each seedling blob
for(i=0; i<objinfo.size(); i++)
{
```

```
            // Compute the junction graph for the seedling blob
            std::vector<Junction*> jlist;
            jg = binskel->CreateJunctionGraph(objinfo[i], jlist);

            // A seedling blob has to have more than 2 junctions
            // to be considered for further processing
            if(jlist.size() > 1)
            {
                  // Find the cotyledon junction.

                  // Find cotyledons and leaves in the current blob.

                  ExImage* cotblobimg = cotimg->Crop(max(0,objinfo[i]->xmin-8),
max(0,objinfo[i]->ymin-8),
                        min(origimg->GetWidth()-1,objinfo[i]->xmax+8), min(origimg-
>GetHeight()-1,objinfo[i]->ymax+8));

                  ExImage* coatblobimg = coatimg->Crop(max(0,objinfo[i]->xmin-8),
max(0,objinfo[i]->ymin-8),
                        min(origimg->GetWidth()-1,objinfo[i]->xmax+8), min(origimg-
>GetHeight()-1,objinfo[i]->ymax+8));

                  // Throw out blobs of size less than c_mincotsize
                  // cotinfo contains information for all extracted cotyledon
                  // blobs greater than or equal to c_mincotsize

                  BinaryObjects* cotobjs = cotblobimg->GetObjects();
                  delete cotblobimg;
                  cotobjs->FilterOut(c_mincotsize);
                  std::vector<ObjectInfo*> cotinfo = cotobjs->GetObjectInfo();

                  // Throw out blobs of size less than c_mincoatsize
                  // coatinfo contains information for all extracted seed coat
                  // blobs greater than or equal to c_mincoatsize

                  BinaryObjects* coatobjs = coatblobimg->GetObjects();
                  delete coatblobimg;
                  coatobjs->FilterOut(c_mincoatsize);
                  std::vector<ObjectInfo*> coatinfo = coatobjs->GetObjectInfo();

                  // coatindx holds id of seed coat junctions
                  std::vector<int> coatindx;

                  // Find the junction (in junction graph) that corresponds to seed coats
                  for(int c=0; c<coatinfo.size(); c++)
                  {
                        // Since objects were extracted from a cropped image, add offset
                        int coatx = max(0,objinfo[i]->xmin-8)+(coatinfo[c]->xmax +
coatinfo[c]->xmin)/2;
                        int coaty = max(0,objinfo[i]->ymin-8)+(coatinfo[c]->ymax +
coatinfo[c]->ymin)/2;

                        float mindist = 1000000.0f;
                        int minindx = -1;

                        // Loop through each junction in the junction graph to
                        // find the closest junction to the current seed coat
```

```
                        for(int j=0; j<jlist.size(); j++)
                        {
                              Junction* junc = jlist[j];
                              int dist = (coatx - junc->m_x) * (coatx - junc->m_x) +
(coaty - junc->m_y) * (coaty - junc->m_y);
                              // the closest junction must be less than 10 pixels away
from the seed coat blob
                              if((dist < mindist) && (dist < 10*10))
                              {
                                    mindist = dist;
                                    minindx = j;
                              }
                        }
                        // if such a seed coat is found, add it to coatindx
                        if(minindx >= 0)
                        {
                              coatindx.push_back(minindx);
                        }
                  }


                  // cotindx holds id of seed coat junctions.
                  std::vector<int> cotindx;

                  // Find the junction that corresponds to cotyledons.
                  for(c=0; c<cotinfo.size(); c++)
                  {
                        cotindx.push_back(-1);

                        // Since objects were extracted from a cropped image, add offset.
                        int cotx = max(0,objinfo[i]->xmin-8)+(cotinfo[c]->xmax +
cotinfo[c]->xmin)/2;
                        int coty = max(0,objinfo[i]->ymin-8)+(cotinfo[c]->ymax +
cotinfo[c]->ymin)/2;

                        TRACE3("blob %d: cotx=%d, coty=%d\n", i, cotx, coty);

                        float mindist = 1000000.0f;

                        // Loop through each junction in the junction graph to
                        // find the closest junction to the current cotyledon
                        for(int j=0; j<jlist.size(); j++)
                        {
                              Junction* junc = jlist[j];
                              int dist = (cotx - junc->m_x) * (cotx - junc->m_x) +  (coty
- junc->m_y) * (coty - junc->m_y);
                              if(dist < mindist)
                              {
                                    mindist = dist;
                                    cotindx[c] = j;
                              }
                        }
                  }

                  // Each seed coat that is close enough to a cotyledon is merged to the
cotyledon
                  for(int cot=0; cot<cotindx.size(); cot++)
```

```
		{
			for(int coat=0; coat<coatindx.size(); coat++)
			{
				// if junctions are within 10 pixel distance, then drop the
coat junction.
				float sqdist = pow(jlist[cotindx[cot]]->m_x -
jlist[coatindx[coat]]->m_x, 2) +
					pow(jlist[cotindx[cot]]->m_y - jlist[coatindx[coat]]-
>m_y, 2);
				if(sqdist < 10.0f*10.0f)
				{
					coatindx.erase(coatindx.begin()+coat);
					coat--;
				}
			}
		}

		// Free memory
		for(int obj=0; obj<cotinfo.size(); obj++)
			delete cotinfo[obj];

		// Free memory
		for(obj=0; obj<coatinfo.size(); obj++)
			delete coatinfo[obj];

		// primaryAxes holds a primary axis for each detected seedling
		// in the current seedling blob
		std::vector<std::vector<int> > primaryAxes;

		// If no cotyledon/seed coat was detected,
		// assume the seedling blob contains only one seedling
		if((cotindx.size() == 0) && (coatindx.size() == 0))
		{
			// Find the longest shortest path to find the primary axis
			float maxdist = 0.0f;
			int maxindx1 = -1, maxindx2 = -1;
			for(int k=0; k<jg->GetSize(); k++)
			{
				std::vector<float> dist = spf.FindShortestPathDistance(*jg,
k);

				for(int m=0; m<dist.size(); m++)
				{
					if(maxdist < dist[m])
					{
						maxdist = dist[m];
						maxindx1 = k;
						maxindx2 = m;
					}
				}
			}
			ASSERT((maxindx1 != -1) && (maxindx2 != -1));
			// Since no information is given on which end junction belongs
			// to a cotyledon, find out which index is top/bottom.
			// Assume the top junction belongs to a cotyledon.
			int y1 = jlist[maxindx1]->m_y;
			int y2 = jlist[maxindx2]->m_y;
```

```
                        if(y1 > y2)
                        {
                              int temp = maxindx1;
                              maxindx1 = maxindx2;
                              maxindx2 = temp;
                        }

                        primaryAxes.push_back(spf.FindShortestPath(*jg, maxindx1,
maxindx2));
                  }
                  // If a cotyledon was detected but no seed coat, find the shortest
                  // path from the cotyledon junction to every other junciton,
                  // and take the longest path as the primary axis
                  else if((cotindx.size() == 1)&&(coatindx.size()==0))
                  {
                              std::vector<float> dist = spf.FindShortestPathDistance(*jg,
cotindx[0]);

                              int maxindx = -1;
                            float maxdist = -1.0f;

                              for(int m=0; m<dist.size(); m++)
                              {
                                    if(maxdist < dist[m])
                                    {
                                          maxdist = dist[m];
                                          maxindx = m;
                                    }
                              }

                              ASSERT(maxindx >= 0);

                        int y1 = jlist[cotindx[0]]->m_y;
                        int y2 = jlist[maxindx]->m_y;

                        int indx1, indx2;

                        if(y1 > y2)
                        {
                              indx1 = maxindx;
                              indx2 = cotindx[0];
                        }
                        else
                        {
                              indx1 = cotindx[0];
                              indx2 = maxindx;
                        }

                        primaryAxes.push_back(spf.FindShortestPath(*jg, indx1, indx2));
                  }
                  // If there is one seed coat and no cotyledon, assume one seedling
                  // within the blob
                  else if((coatindx.size() == 1)&&(cotindx.size()==0))
                  {
                        // Find the longest shortest path.
                              std::vector<float> dist = spf.FindShortestPathDistance(*jg,
coatindx[0]);
```

```
                        int maxindx = -1;
                    float maxdist = -1.0f;

                        for(int m=0; m<dist.size(); m++)
                        {
                              if(maxdist < dist[m])
                              {
                                    maxdist = dist[m];
                                    maxindx = m;
                              }
                        }
                        ASSERT(maxindx >= 0);

                  int y1 = jlist[coatindx[0]]->m_y;
                  int y2 = jlist[maxindx]->m_y;

                  int indx1, indx2;

                  if(y1 > y2)
                  {
                        indx1 = maxindx;
                        indx2 = coatindx[0];
                  }
                  else
                  {
                        indx1 = coatindx[0];
                        indx2 = maxindx;
                  }

                  primaryAxes.push_back(spf.FindShortestPath(*jg, indx1, indx2));
            }
            // This is the case where there are multiple cotyledons in the blob
            else if(coatindx.size() + cotindx.size() > 1)
            {
                  std::vector<int> startjuncs;
                  for(int c=0; c<coatindx.size(); c++)
                        startjuncs.push_back(coatindx[c]);
                  for(c=0; c<cotindx.size(); c++)
                        startjuncs.push_back(cotindx[c]);
                  // Obtain primary axis for each seedling in the blob
                  primaryAxes = SeparateSeedlings(jg, startjuncs, c_maxIteration,
c_lengthW, c_angleW, c_unusedW, c_initialTemperature, c_temperatureConst,
c_minEdgeLength, c_separationW);
            }

            // Perform hypocotyl/radicle separation on each seedling found in the
blob
            // The first junction encountered while traversing the primary path
from the
            // cotyledon junction that separates the seedling into hypocotyl and
radicle such that
            // hypocotyl : radicle length ratio is no less than 0.15
            // is the separation point

            for(int s=0; s<primaryAxes.size(); s++)
            {
```

```
                        if(primaryAxes[s].size() > 1)
                        {
                              int k = 1;

                              SeedlingInfo* sinfo = new SeedlingInfo;
                              sinfo->hypocotyl_length = jg->GetEdge(primaryAxes[s][k-1],
primaryAxes[s][k])->m_length;
                              sinfo->radicle_length = 0;

                              for(int j=1; j<primaryAxes[s].size()-1; j++)
                              {
                                    sinfo->radicle_length += jg-
>GetEdge(primaryAxes[s][j], primaryAxes[s][j+1])->m_length;
                              }

                              // Check to see if hypocotyl/radicle ratio is not so
extreme.
                              // If so, extend hypocotyl and shorten radicle.
                              k++;
                              while(((float)sinfo->hypocotyl_length / sinfo-
>radicle_length < 0.15) && (primaryAxes[s].size() > k))
                              {
                                    sinfo->hypocotyl_length += jg-
>GetEdge(primaryAxes[s][k-1], primaryAxes[s][k])->m_length;
                                    sinfo->radicle_length -= jg-
>GetEdge(primaryAxes[s][k-1], primaryAxes[s][k])->m_length;
                                    k++;
                              }

                              sinfo->hyporad_separation = k - 1;
                              sinfo->primary_axis = primaryAxes[s];
                              sinfo->junction_graph = jg;

                              // Compute the bounding box for the seedling

                              int xmin = 1000000, xmax = -1, ymin = 1000000, ymax = -1;
                              for(int m=0; m<primaryAxes[s].size()-1; m++)
                              {
                                    Edge* edge = jg->GetEdge(primaryAxes[s][m],
primaryAxes[s][m+1]);
                                    if(edge->m_xmin < xmin)
                                          xmin = edge->m_xmin;
                                    if(edge->m_xmax > xmax)
                                          xmax = edge->m_xmax;
                                    if(edge->m_ymin < ymin)
                                          ymin = edge->m_ymin;
                                    if(edge->m_ymax > ymax)
                                          ymax = edge->m_ymax;
                              }

                              sinfo->topleft.x = xmin;
                              sinfo->topleft.y = ymin;
                              sinfo->bottomright.x = xmax;
                              sinfo->bottomright.y = ymax;

                              seedlinginfo.push_back(sinfo);
```

```
                        added = true;
                    }
                } // if there is one cotyledon in the blob
            } // if there are more than one junctions
            if(!added)
                delete jg;
            else
                added = false;
        }

        // In case cotyledon/hypocotyl separation was undetected,
        // assume separation point at the mean ratio.

        for(int i=0; i<seedlinginfo.size(); i++)
        {
            if(seedlinginfo[i]->hyporad_separation == 0)
            {
                seedlinginfo[i]->useAverageSeparation = true;
            }
        }

        // Free memory.

        delete cotimg;
        delete coatimg;

        return seedlinginfo;
    }


// Separate seedlings using simulated annealing.
std::vector<std::vector<int> > SeedlingAnalyzer::SeparateSeedlings(ConnectivityGraph<Edge
>* jgraph, std::vector<int> startjunc,
int loopmax, float lengthW, float angleW, float unusedW, float temperature, float
temperatureConst, float minEdgeLength, float separationW)
{
        // start contains the ID of junctions that are preassigned to seedlings.
        // start.size() is the number of seedlings assigned initially.

        // Keep track of the path for each seedling.
        std::vector<std::vector<int> > paths;

        // Keep track of the length for the paths.
        std::vector<float> pathlength;

        // Keep track of the last "valid" end angle taken for each seedling.
        std::vector<std::vector<float> > endAngle;

        // Keep track of hypocotyl/radicle separation
        std::vector<int> RHseparation;
        std::vector<int> RHseparation2;
        std::vector<float> separationlen;

        // Initialize data structures by starting out with a single
        // junction for each seedling
        for(int i=0; i<startjunc.size(); i++)
        {
            std::vector<int> path;
```

```
        paths.push_back(path);
        paths[i].push_back(startjunc[i]);

        std::vector<float> list;
        endAngle.push_back(list);

        pathlength.push_back(0.0f);
        RHseparation.push_back(-1);  // Initially, paths don't have RH separation
        RHseparation2.push_back(-1);
        separationlen.push_back(0.0f);
    }


    // Keep track of which edge is occupied

    std::vector<std::set<int> > edgeOccupation;
    for(i=0; i<jgraph->GetNumEdges(); i++)
    {
        std::set<int> edgeset;
        edgeOccupation.push_back(edgeset);
    }

    // Set initial temperature for annealing.

    float energy = 0.0f;

    // Compute the energy of the configuration.
    // Since no edges are occupied by seedlings yet,
    // the energy is the sum of penalties for
    // unoccupied edges.
    for(i=0; i<jgraph->GetSize(); i++)
    {
        for(int j=i+1; j<jgraph->GetSize(); j++)
        {
            if(jgraph->IsEdge(i, j))
                energy += unusedW * jgraph->GetEdge(i, j)->m_length;
        }
    }

    // add energy for not having hypocotyl/radicle separation.
    energy += separationW * startjunc.size();

    for(i=0; i<loopmax; i++)
    {
        // Choose which seedling to update.
        int seedID = (int)(GETRAND*(double)startjunc.size());
        int endjunc = paths[seedID].back();

        bool changeAngle = false;  // Whether to change the last "valid" angle or
not.


        // Extend end or remove end.
        // Choose an edge by throwing a die.

        // Copy all neighboring junctions to neighbors.
        std::vector<int> neighbors;
        for(int j=0; j<jgraph->GetSize(); j++)
```

```
        {
            if(jgraph->IsEdge(endjunc, j))
                neighbors.push_back(j);
        }
        int choice = neighbors[(int)(GETRAND*(double)neighbors.size())];

        // Try to remove edge if choice is the junction one before the
        // current junction.
        if((paths[seedID].size() > 1) && (choice == *(paths[seedID].end()-2)))
        {
            // Remove edge --- compute delta energy.
            // *-------> ... *-------->*
            // start        choice    end

            float deltaEnergy;
            float edgeLength = jgraph->GetEdge(choice, endjunc)->m_length;

            // Removing hypo/rad separation increases energy
            if(RHseparation2[seedID] == choice)
            {
                deltaEnergy += separationW;
            }


            // If the edge is sufficiently long, subtract energy for the angle
            if(edgeLength > minEdgeLength)
            {
                if(endAngle[seedID].size() > 1)
                {
                    float newangle = jgraph->GetEdge(choice, endjunc)-
>GetAngle(choice);
                    float angle = ComputeAngle(*(endAngle[seedID].end()-2),
newangle);
                    deltaEnergy -= angleW * angle * angle;
                }
                changeAngle = true;
            }

            // If the edge is no longer occupied after removal, increase
            // energy.
            if((edgeOccupation[jgraph->GetEdgeID(choice, endjunc)].size()==1) &&
               (*edgeOccupation[jgraph->GetEdgeID(choice, endjunc)].begin() ==
seedID))
                deltaEnergy += unusedW * edgeLength;

            if((deltaEnergy < 0.0f) || (exp(-deltaEnergy/(temperatureConst*
temperature))) > GETRAND)
            {
                // Remove the end junction.
                ASSERT(!paths[seedID].empty());
                edgeOccupation[jgraph->GetEdgeID(choice, endjunc)].erase(seedID);
                paths[seedID].pop_back();

                if(changeAngle)
                {
                    ASSERT(endAngle.size() > 0);
                    endAngle[seedID].pop_back();
```

```
                        }

                        // if the chosen junction was the hypo/rad separation for
                        // the seedling, unmark the separation.
                        if(RHseparation[seedID] == choice)
                        {
                              RHseparation[seedID] = -1;
                        }

                        if(RHseparation2[seedID] == choice)
                        {
                              RHseparation2[seedID] = -1;
                        }

                        pathlength[seedID] -= edgeLength;

                        energy += deltaEnergy;
                  }
            }
            // Add edge if the selected junction is not the end junction itself
            else if((choice != endjunc) && (edgeOccupation[jgraph->GetEdgeID(endjunc,
choice)].find(seedID) == edgeOccupation[jgraph->GetEdgeID(endjunc, choice)].end()))
            {
                  // Add edge
                  // *-------> ... *------->*
                  // start        end     choice

                  ASSERT(jgraph->IsEdge(endjunc, choice));

                  bool newseparation = false;  // Whether choice adds a hypo/rad
separation or not.
                  bool separationcomplete = false;

                  float deltaEnergy;
                  float edgeLength = jgraph->GetEdge(endjunc, choice)->m_length;
                  // If the edge is not already occupied, the energy goes down.
                  if(edgeOccupation[jgraph->GetEdgeID(endjunc, choice)].empty())
                        deltaEnergy -= unusedW * edgeLength;

                  int numneighbors = 0;
                  // See if choice is a separation point by checking for a
                  // neighbor edge that is short (and has a degree one at the other
end??)
                  for(int j=0; j<jgraph->GetSize(); j++)
                  {
                        if(jgraph->IsEdge(choice, j))
                        {
                              numneighbors++;
                        }
                  }

                  // Mark if this is a new hypo/rad separation.
                  if(RHseparation[seedID] == -1)
                  {
                        if((numneighbors > 2) && (pathlength[seedID] > 20.0f))
                        {
                              newseparation = true;
```

```
                        }
                  }
                  else if((RHseparation2[seedID] == -1) && (pathlength[seedID] -
separationlen[seedID] > 20.0f))
                  {
                        deltaEnergy -= separationW;
                        separationcomplete = true;
                  }


                  if(edgeLength > minEdgeLength)
                  {
                        if(!endAngle[seedID].empty())
                        {
                              float newangle = jgraph->GetEdge(endjunc, choice)-
>GetAngle(endjunc);
                              float angle = ComputeAngle(endAngle[seedID].front(),
newangle);
                              deltaEnergy += angleW * angle * angle;
                              // egraph->GetEdge(jgraph-
>GetEdgeID(paths[seedID][paths[seedID].size()-2], endjunc), jgraph->GetEdgeID(endjunc,
choice));
                        }
                        changeAngle = true;
                  }

                  if((deltaEnergy < 0.0f) || (exp(-deltaEnergy/(temperatureConst*
temperature))) > GETRAND)
                  {
                        // mark that this seedling has occupied the edge
                        edgeOccupation[jgraph->GetEdgeID(endjunc,
choice)].insert(seedID);
                        ASSERT(edgeOccupation[jgraph->GetEdgeID(endjunc, choice)].size()
== startjunc.size());
                        paths[seedID].push_back(choice);

                        if(changeAngle)
                              endAngle[seedID].push_back(jgraph->GetEdge(endjunc,
choice)->GetAngle(choice));

                        pathlength[seedID] += edgeLength;

                        if(newseparation)
                        {
                              RHseparation[seedID] = choice;
                              separationlen[seedID] = pathlength[seedID];
                        }

                        if(separationcomplete)
                              RHseparation2[seedID] = choice;

                        energy += deltaEnergy;
                  }
            }

            // Adjust temperature.
            temperature *= 0.99f;
```

```
	} // Simulated annealing loop.

	return paths;
}

float SeedlingAnalyzer::ComputeAngle(float lastAngle, float newAngle)
{
	// Compute the angle between lastAngle and newAngle.

	float angle = fabs(newAngle - lastAngle);
	if(angle > PI)
		angle = DPI - angle;

	return (PI-angle);
}
```

```
// File: Image.cpp
// Author: Yusaku Sako
// Date: 07/11/99

#include "stdafx.h"  // for Windows
#include "Image.h"
#include "ImageThinning.h"  // for thinning a binary image
#include "ConnectivityGraph.h"
#include "Const.h"
#include <math.h>

#ifndef ROUND
#define ROUND(x) ((int)(x+0.5))
#endif


ExImage* ExImage::Crop(int lx, int ly, int rx, int ry)
{
	Image* dupimage = duplicate_Image(m_image);
	Image* newimage = ::crop(dupimage, ly, lx, ry-ly+1, rx-lx+1);

	ExImage* returnimage = new ExImage;
	returnimage->m_image = newimage;

	return returnimage;


ExImage* ExImage::TransformToHSV(int numlevels)

	ExImage* newimg = (ExImage*)GetCopy();

	float norm[3];
	norm[0] = numlevels;  norm[1] = numlevels;  norm[2] = numlevels;
	newimg->m_image = colorxform(newimg->m_image, HSV, norm, NULL, 1);

	return newimg;
}

ExImage* ExImage::Get2DHistogram(int xcolor, int ycolor, int size)
{
	ExImage* histImg = new ExImage;
	histImg->m_image = new_Image(PGM, GRAY_SCALE, 1, size,
		size, CVIP_BYTE, REAL);

	const int HIGH = 255;
	const int BIAS = 128;
	unsigned char ** xpix = (unsigned char**)m_image->image_ptr[xcolor]->rptr;
	unsigned char ** ypix = (unsigned char**)m_image->image_ptr[ycolor]->rptr;
	unsigned char ** hpix = (unsigned char**)histImg->m_image->image_ptr[0]->rptr;

	int kx, ky, hx, hy, max, kk;

	// Clear image.
	for(int i=0; i<size; i++)
		for(int j=0; j<size; j++)
		{
			hpix[i][j] = (unsigned char)0;
```

```
            }

      max = 0;
      kx = 1;
      ky = 1;
      for(i=0; i<size; i++)
      {
            for(int j=0; j<size; j++)
            {
                  hy = (HIGH - ypix[i][j])/ky;
                  hx = (xpix[i][j])/kx;
                  if(hpix[hy][hx] < HIGH)
                        hpix[hy][hx]++;
                  if(max < hpix[hy][hx])
                        max = hpix[hy][hx];
            }
      }
      for(i=0; i<size; i++)
      {
            for(int j=0; j<size; j++)
            {
                  if(hpix[i][j] !=0)
                  {
                        kk = hpix[i][j]*HIGH/max+BIAS;
                        if(kk > HIGH)
                              hpix[i][j] = (unsigned char)HIGH;
                        else
                              hpix[i][j] = (unsigned char)kk;
                  }
            }
      }
      return histImg;


ExImage* ExImage::Threshold(unsigned char thresh)
{
      ExImage *returnImage = new ExImage;

      returnImage->m_image = threshold_segment(m_image, thresh, CVIP_NO);
 /*
 <inputImage> - pointer to Image structure
 <threshval> - threshold value
 <thresh_inbyte>
          - CVIP_NO apply threshval directly to image data;
          - CVIP_YES  threshval is CVIP_BYTE range; remap to image
                data range before thresholding.
 */
      return returnImage;
}

ExImage* ExImage::Threshold(int rlow, int rhigh, int glow, int ghigh, int blow, int
bhigh)
{
      ExImage* threshImg = new ExImage;
      threshImg->m_image = new_Image(PGM, GRAY_SCALE, 1, getNoOfRows_Image(m_image),
            getNoOfCols_Image(m_image), CVIP_BYTE, REAL);
```

```
	unsigned char** origR = (unsigned char**)m_image->image_ptr[0]->rptr;
	unsigned char** origG = (unsigned char**)m_image->image_ptr[1]->rptr;
	unsigned char** origB = (unsigned char**)m_image->image_ptr[2]->rptr;

	unsigned char** dest = (unsigned char**)threshImg->m_image->image_ptr[0]->rptr;

	for(int y=0; y<getNoOfRows_Image(m_image); y++)
		for(int x=0; x<getNoOfCols_Image(m_image); x++)
		{
			bool flag = true;

			if(origR[y][x] < rlow || origR[y][x] > rhigh)
				flag = false;
			if(origG[y][x] < glow || origG[y][x] > ghigh)
				flag = false;
			if(origB[y][x] < blow || origB[y][x] > bhigh)
				flag = false;

			if(flag)
				dest[y][x] = 255;
			else
				dest[y][x] = 0;
		}

	return threshImg;


ExImage* ExImage::Threshold(ThreshParams thresh)
{
	return Threshold(thresh.rmin, thresh.rmax, thresh.gmin, thresh.gmax, thresh.bmin,
thresh.bmax);


// Perform Thresholding segmentation based on histogram
ExImage* ExImage::HistogramThreshold()
{
	ExImage *returnImage = new ExImage;
	returnImage->m_image = hist_thresh_gray(m_image);
	return returnImage;
}

ExImage* ExImage::HistogramEqualization()
{
	ExImage *returnImage = new ExImage;
	Image* temp;

	temp = histeq(m_image, 0);
	//temp = histeq(temp, 1);
	//temp = histeq(temp, 2);
	returnImage->m_image = temp;
	returnImage->m_image = remap_Image(temp, CVIP_BYTE, 0, 255);

	return returnImage;
}

ExImage* ExImage::EdgeDetect(int type)
{
```

```
	ExImage *returnImage = new ExImage;
	returnImage->m_image = edge_detect_setup(m_image, 1);
	return returnImage;
}

// Perform skeletonization on the image.
ExImage* ExImage::Thin()
{
	ExImage *returnImage = new ExImage;
	returnImage->m_image = ::ImageThinning(m_image);
	return returnImage;
}

ExImage* ExImage::MorphOpen(int kerneltype, int kernelsize,
		int kernelheight, int kernelwidth)
{
	ExImage *returnImage = new ExImage;
	returnImage->m_image = ::MorphOpen(m_image, kerneltype, kernelsize, kernelheight,
kernelwidth);
	// DEBUG
	CString msg;
	msg.Format("Image params: width=%d height=%d bands=%d colorspace=%d", returnImage-
>m_image->image_ptr[0]->cols, returnImage->m_image->image_ptr[0]->rows, returnImage-
>m_image->bands, returnImage->m_image->color_space);
	//AfxMessageBox(msg);
	return returnImage;


ExImage* ExImage::MorphClose(int kerneltype, int kernelsize,
		int kernelheight, int kernelwidth)

	ExImage *returnImage = new ExImage;
	returnImage->m_image = ::MorphClose(m_image, kerneltype, kernelsize, kernelheight,
kernelwidth);
	return returnImage;
}

ExImage* ExImage::MorphDilate(int kerneltype, int kernelsize,
		int kernelheight, int kernelwidth)
{
	ExImage *returnImage = new ExImage;
	returnImage->m_image = ::MorphDilate(m_image, kerneltype, kernelsize, kernelheight,
kernelwidth);
	return returnImage;
}

ExImage* ExImage::MorphErode(int kerneltype, int kernelsize,
		int kernelheight, int kernelwidth)
{
	ExImage *returnImage = new ExImage;
	returnImage->m_image = ::MorphErode(m_image, kerneltype, kernelsize, kernelheight,
kernelwidth);
	return returnImage;
}

// Convert a color image to gray scale based on luminance.
ExImage* ExImage::ConvertToGrayscale(int maxvalue)
```

```
{
      ExImage *returnImage = new ExImage;
      returnImage->m_image = ::CVIPluminance(m_image, maxvalue,
     CVIP_YES, CVIP_NO);
      return returnImage;
}

// Perform median filter on the image.
ExImage* ExImage::MedianFilter(int kernelsize)
{
      ExImage *returnImage = new ExImage;
      returnImage->m_image = ::median_filter(m_image, kernelsize);
      return returnImage;
}

ExImage* ExImage::CutOut(int lx, int ly, int rx, int ry)
{
      ASSERT(lx>=0);
      ASSERT(ly>=0);
      ASSERT(rx<getNoOfCols_Image(m_image));
      ASSERT(ry<getNoOfRows_Image(m_image));

      Image* dupimg = duplicate_Image(m_image);
      unsigned char** pix = (unsigned char**)dupimg->image_ptr[0]->rptr;

      for(int y=ly; y<=ry; y++)
      {
            for(int x=lx; x<=rx; x++)
            {
                  pix[y][x] = 0U;
            }
      }

      ExImage *returnImage = new ExImage;
      returnImage->m_image = dupimg;
      return returnImage;
}

ExImage* ExImage::Minus(ExImage* inimage, int lx, int ly, int rx, int ry)
{
      Image* diffimg = duplicate_Image(m_image);

      unsigned char** pix1 = (unsigned char**)diffimg->image_ptr[0]->rptr;
      unsigned char** pix2 = (unsigned char**)inimage->m_image->image_ptr[0]->rptr;

      for(int y=ly; y<=ry; y++)
      {
            for(int x=lx; x<=rx; x++)
            {
                  int diff = pix1[y][x] - pix2[y-ly][x-lx];
                  diff = (diff<0) ? 0 : diff;
                  pix1[y][x] = diff;
            }
      }

      ExImage* returnImage = new ExImage;
      returnImage->m_image = diffimg;
```

```
        return returnImage;
}

ExImage* ExImage::Blend(ExImage* inimage, float weight_orig, float weight_in, bool
invert_orig, bool invert_in)
{
        // Image sizes must match.
        ASSERT(getNoOfCols_Image(m_image) == getNoOfCols_Image(inimage->m_image));
        ASSERT(getNoOfRows_Image(m_image) == getNoOfRows_Image(inimage->m_image));
        ASSERT((weight_orig >= 0.0) && ( weight_orig <= 1.0));

        Image* newimage = new_Image(PPM, RGB, 3, getNoOfRows_Image(m_image),
                getNoOfCols_Image(m_image),CVIP_BYTE, REAL);

        int numbands = getNoOfBands_Image(inimage->m_image);

        char **src1R = m_image->image_ptr[0]->rptr;
        char **src2R = inimage->m_image->image_ptr[0]->rptr;
        unsigned char **destR = (unsigned char**)newimage->image_ptr[0]->rptr;
        char **src1G = m_image->image_ptr[1]->rptr;
        char **src2G = (numbands>1) ? inimage->m_image->image_ptr[1]->rptr : inimage-
>m_image->image_ptr[0]->rptr;
        unsigned char **destG = (unsigned char**)newimage->image_ptr[1]->rptr;
        char **src1B = m_image->image_ptr[2]->rptr;
        char **src2B = (numbands>2) ? inimage->m_image->image_ptr[2]->rptr : inimage-
>m_image->image_ptr[0]->rptr;
        unsigned char **destB = (unsigned char**)newimage->image_ptr[2]->rptr;

        unsigned char src1r, src2r, src1g, src2g, src1b, src2b;

        for(int y=0; y<getNoOfRows_Image(m_image); y++)
                for(int x=0; x<getNoOfCols_Image(m_image); x++)
                {
                        if(invert_orig)
                        {
                                src1r = -src1R[y][x]-1;
                                src1g = -src1G[y][x]-1;
                                src1b = -src1B[y][x]-1;
                        }
                        else
                        {
                                src1r = src1R[y][x];
                                src1g = src1G[y][x];
                                src1b = src1B[y][x];
                        }
                        if(invert_in)
                        {
                                src2r = -src2R[y][x]-1;
                                src2g = -src2G[y][x]-1;
                                src2b = -src2B[y][x]-1;
                        }
                        else
                        {
                                src2r = src2R[y][x];
                                src2g = src2G[y][x];
                                src2b = src2B[y][x];
                        }
```

```
                destR[y][x]                                =                (unsigned
char)((weight_orig*(float)(src1r)+weight_in*(float)src2r)/2);
                destG[y][x]                                =                (unsigned
char)((weight_orig*(float)(src1g)+weight_in*(float)src2g)/2);
                destB[y][x]                                =                (unsigned
char)((weight_orig*(float)(src1b)+weight_in*(float)src2b)/2);
            }

      ExImage *ret = new ExImage;
      ret->m_image = newimage;
      return ret;
}

ExImage* ExImage::Mask(ExImage* maskimage, ColorType bgcolor)
{
      ExImage * outimage = (ExImage*)GetCopy();

      unsigned char** mpix = maskimage->GetPixelArray(0);
      unsigned char** srcR = GetPixelArray(0);
      unsigned char** srcG = GetPixelArray(1);
      unsigned char** srcB = GetPixelArray(2);
      unsigned char** dstR = outimage->GetPixelArray(0);
      unsigned char** dstG = outimage->GetPixelArray(1);
      unsigned char** dstB = outimage->GetPixelArray(2);


      for(int y=0; y<maskimage->GetHeight(); y++)
      {
            for(int x=0; x<maskimage->GetWidth(); x++)
            {
                  if(mpix[y][x] > 0)
                  {
                        dstR[y][x] = srcR[y][x];
                        dstG[y][x] = srcG[y][x];
                        dstB[y][x] = srcB[y][x];
                  }
                  else
                  {
                        dstR[y][x] = bgcolor.r;
                        dstG[y][x] = bgcolor.g;
                        dstB[y][x] = bgcolor.b;
                  }
            }
      }

      return outimage;
}

      ObjectList label_Objects2(Image *imageP, Image **labelP, unsigned background);

// Return objects found in the image.
BinaryObjects* ExImage::GetObjects()
{
      Image* labelImage;
      ObjectList objlist;
      objlist = label_Objects2(m_image, &labelImage, 0);
```

```
	// Make sure objlist is not NULL
	if(!objlist)
		objlist = new_LL();
	BinaryObjects *binobjs = new BinaryObjects(objlist, labelImage);
	return binobjs;
}

int ExImage::FindPeakValue(int band)
{
	// Find peak.
	unsigned char **pix = (unsigned char**)m_image->image_ptr[band]->rptr;
	long* histogram = new long[256];

	for(int i=0; i<256; i++)
		histogram[i] = 0;

	for(int y=0; y<GetHeight(); y++)
	{
		for(int x=0; x<GetWidth(); x++)
		{
			histogram[pix[y][x]]++;
		}
	}

	// Find the peak.
	int high = 0;
	for(i=0; i<256; i++)
	{
		if(histogram[i] > histogram[high])
			high = i;
	}

	delete [] histogram;

	return (high);

}

ColorType ExImage::GetAverageColor()
{
	long rsum = 0, gsum = 0, bsum = 0;

	unsigned char** pixR = (unsigned char**)m_image->image_ptr[0]->rptr;
	unsigned char** pixG = (unsigned char**)m_image->image_ptr[1]->rptr;
	unsigned char** pixB = (unsigned char**)m_image->image_ptr[2]->rptr;

	for(int y=0; y<getNoOfRows_Image(m_image); y++)
		for(int x=0; x<getNoOfCols_Image(m_image); x++)
		{
			rsum += pixR[y][x];
			gsum += pixG[y][x];
			bsum += pixB[y][x];
		}

	ColorType color;
	color.r = rsum/(getNoOfRows_Image(m_image)*getNoOfCols_Image(m_image));
	color.g = gsum/(getNoOfRows_Image(m_image)*getNoOfCols_Image(m_image));
```

```
	color.b = bsum/(getNoOfRows_Image(m_image)*getNoOfCols_Image(m_image));

	return color;
}

// Dump properties of all objects.
void BinaryObjects::Dump()
{
	head_LL(m_objectlist);  // set linked list pointer to the head

	for(int i=0; i<size_LL(m_objectlist); i++)
	{
		Object *obj = ((Object *)retrieve_LL(m_objectlist));
		CString msg;
		msg.Format("label=%d; R=%d G=%d B=%d; xmin=%d ymin=%d xmax=%d ymax=%d;\n
eigenratio=%f; orientation=%f; horizontal cog=%f vertical cog=%f; area=%f",
			obj->label, obj->pixel.r, obj->pixel.g, obj->pixel.b, obj->x_min, obj-
>y_min,   obj->x_max,   obj->y_max,   obj->prop.eig_ratio,   obj->prop.orientation,   obj-
>prop.h_cog, obj->prop.v_cog, obj->prop.area);
		AfxMessageBox(msg);
		next_LL(m_objectlist);
	}



int _cdecl MatchInt(void* content, void* lookforP)
{
	return(*((int*)content) == *((int*)lookforP));
}

// Remove objects whose sizes are equal to or less than minarea.
void BinaryObjects::FilterOut(int minarea)
{
	if(m_objectlist->listlength == 0)
		return;

	getProp_Objects(m_objectlist, m_labelImage);

	head_LL(m_objectlist);
	previous_LL(m_objectlist);

	//int** pix = (int**)m_labelImage->image_ptr[0]->rptr;

	for(;;)
	{
		Object* obj = ((Object *)retrieveNext_LL(m_objectlist));

		if(obj->prop.area < minarea)
		{
			// Erase pixels belonging to this object from the label image
			for(int y=obj->y_min; y<=obj->y_max; y++)
				for(int x=obj->x_min; x<=obj->x_max; x++)
				{
					if(((int*)m_labelImage->image_ptr[0]->rptr[y])[x]   ==   obj-
>label)
					{
						((int*)m_labelImage->image_ptr[0]->rptr[y])[x] = 0;
```

```
                        }
                    }

                removenext_LL(m_objectlist);
            }
            else
                next_LL(m_objectlist);

            if(istail_LL(m_objectlist))
                break;
        }
}

ExImage* BinaryObjects::CreateContourImage()
{
        Image* returnimg;
        returnimg = new_Image(PBM, BINARY, 1, getNoOfRows_Image(m_labelImage),
                getNoOfCols_Image(m_labelImage), CVIP_BYTE, REAL);
        /*
        returnimg = new_Image(TIF, BINARY,
       1, getNoOfRows_Image(m_labelImage), getNoOfCols_Image(m_labelImage),
         CVIP_BYTE, REAL);
        */
    // Extract chain code from each object, and draw them onto returnimg.

        // dump labelled image

        CString msg;
        /*
        msg.Format("#   bands=%d   width=%d   height=%d   format=%d   type=%d,   space=%d",
m_labelImage->bands,
                getNoOfCols_Image(m_labelImage),              getNoOfRows_Image(m_labelImage),
m_labelImage->image_format,
                m_labelImage->image_ptr[0]->data_type, m_labelImage->color_space);
        AfxMessageBox(msg);
        msg.Format("# bands=%d width=%d height=%d format=%d type=%d, space=%d", returnimg-
>bands,
                getNoOfCols_Image(returnimg),     getNoOfRows_Image(returnimg),     returnimg-
>image_format,
                returnimg->image_ptr[0]->data_type, returnimg->color_space);
        AfxMessageBox(msg);
        */

    head_LL(m_objectlist);

    for(;;)
    {
            int ray_x;

        Object *obj = ((Object*)retrieve_LL(m_objectlist));
            //shootRay(m_labelImage, obj->label, &ray_x, &ray_y, obj->x_min, obj->y_min,
obj->x_max, obj->y_max);
            for(int c=obj->x_min; c<=obj->x_max; c++)
            {
                if(((int*)m_labelImage->image_ptr[0]->rptr[obj->y_min])[c]    ==    obj-
>label)
                {
```

```
                        ray_x = c;
                        break;
                    }
                }
            ChainCode* cc = new_ChainCode(obj->y_min, ray_x, obj->label);
            if(build_LineChainCode(cc, m_labelImage, obj->x_min, obj->y_min, obj->x_max, obj->y_max)==0)
                AfxMessageBox("Error building chain code!");
            draw_ChainCode(cc, returnimg);
            if(istail_LL(m_objectlist))
                break;
            next_LL(m_objectlist);
        }

        ExImage* newimg = new ExImage;
        newimg->m_image = returnimg;

        return newimg;
}

ExImage* BinaryObjects::CreateImage()
{
        Image* returnimg;
        returnimg = new_Image(PGM, GRAY_SCALE, 1, getNoOfRows_Image(m_labelImage),
                getNoOfCols_Image(m_labelImage), CVIP_BYTE, REAL);

        for(int y=0; y<getNoOfRows_Image(m_labelImage); y++)
                for(int x=0; x<getNoOfCols_Image(m_labelImage); x++)
                {
                        if(((int*)m_labelImage->image_ptr[0]->rptr[y])[x] > 0)
                                returnimg->image_ptr[0]->rptr[y][x] = 255U;
                        else
                                returnimg->image_ptr[0]->rptr[y][x] = 0;
                }

        ExImage* newimg = new ExImage;
        newimg->m_image = returnimg;

        return newimg;
}

ConnectivityGraph<Edge    *>*    BinaryObjects::CreateJunctionGraph(ObjectInfo*    objinfo,
std::vector<Junction*>& junclist /* output */)
{
        // Find all junctions.
        junclist = ComputeJunctions(objinfo);

        TRACE("There are %d pre-junctions\n", junclist.size());

        ConnectivityGraph<Edge    *>*    cg    =    new    ConnectivityGraph<Edge    *>(false,
junclist.size());

        //int *degreelist = new int[junclist.size()];

        // Copy degrees.
        //for(int i=0; junclist.size(); i++)
        //{
```

```
	//	degreelist[i] = junclist[i]->degree;
	//}

	// Connect nodes.
	for(int i=0; i<junclist.size(); i++)
	{
		int c = 0;
		float oldlen;
		while(junclist[i]->m_neighborPixels.size() != 0)
		{
			Edge *edge = new Edge;
			TRACE("j[%d]'s next neighbor is %d\n", i, *(junclist[i]->m_neighborPixels.begin()));
			int nextjunc = FindNextJunction(junclist, *(junclist[i]->m_neighborPixels.begin()), i, *edge);
			// If nextjunc is i itself, then this is a terminal loop.
			// Simply ignore it.
			if(nextjunc != i)
			{
				TRACE("Inserting edge (%d, %d)\n", i, nextjunc);

				// If there is already an edge (i, nextjunc), ignore the new edge.
				if(!cg->IsEdge(i, nextjunc))
				{
					oldlen = edge->m_length;
					cg->InsertEdge(i, nextjunc, edge);
				}
				else if(edge->m_length < oldlen)
				{
					cg->DeleteEdge(i, nextjunc);
					cg->InsertEdge(i, nextjunc, edge);
				}

				// Erase the neighbor that leads to nextjunc and
				// erase the neighbor that leads to i so as to avoid
				// duplicate processing of the same edge.
				TRACE("Erasing neighbor %d from j[%d] and %d from j[%d]\n", edge->GetNeighbor(i), i, edge->GetNeighbor(nextjunc), nextjunc);
				junclist[i]->m_neighborPixels.erase(edge->GetNeighbor(i));
				junclist[nextjunc]->m_neighborPixels.erase(edge->GetNeighbor(nextjunc));
				//degreelist[i]--;
				c++;
			}
			else
			{
				// Erase the neighbor pixel that leads to the loop
				// In case of loop, edge->GetNeighbor(0) returns the first neighbor
				// and edge->GetNeighbor(1) returns the last neighbor
				junclist[i]->m_neighborPixels.erase(edge->GetNeighbor(0));
				junclist[i]->m_neighborPixels.erase(edge->GetNeighbor(1));
			}
		}
	}
```

```
	TRACE("There are %d post edges\n", cg->GetNumEdges());

	return cg;
}

int BinaryObjects::FindNextJunction(std::vector<Junction *> junclist, int neighbor, int
start, Edge& edge)
{
	// FOR DEBUG
	FILE* out = fopen("junction.txt", "w");

	int** pix = (int**)m_labelImage->image_ptr[0]->rptr;

	int deg;

	int x = junclist[start]->m_x;
	int y = junclist[start]->m_y;

	edge.m_junc1 = start;

	// Keep track of all coordinates.
	edge.m_xarray.clear();
	edge.m_yarray.clear();

	edge.m_xarray.push_back(x);
	edge.m_yarray.push_back(y);

	edge.m_xmin = 1000000;
	edge.m_xmax = -1;
	edge.m_ymin = 1000000;
	edge.m_ymax = -1;

	// FOR DEBUG
	fprintf(out, "finding the next junction for junction %d\n", start);

	do
	{
		fprintf(out, "neighbor=%d\n", neighbor);
		switch(neighbor)
		{
		case 0:
			x = x+1;
			y = y;
			edge.m_length += 1.0f;
			break;
		case 1:
			x = x+1;
			y = y+1;
			edge.m_length += 1.41421356f;
			break;
		case 2:
			x = x;
			y = y+1;
			edge.m_length += 1.0f;
			break;
		case 3:
			x = x-1;
```

```
            y = y+1;
            edge.m_length += 1.41421356f;
            break;
      case 4:
            x = x-1;
            y = y;
            edge.m_length += 1.0f;
            break;
      case 5:
            x = x-1;
            y = y-1;
            edge.m_length += 1.41421356f;
            break;
      case 6:
            X = X;
            y = y-1;
            edge.m_length += 1.0f;
            break;
      case 7:
            x = x+1;
            y = y-1;
            edge.m_length += 1.41421356f;
            break;
      }

      if(x < edge.m_xmin)
            edge.m_xmin = x;
      if(x > edge.m_xmax)
            edge.m_xmax = x;
      if(y < edge.m_ymin)
            edge.m_ymin = y;
      if(y > edge.m_ymax)
            edge.m_ymax = y;

      edge.m_xarray.push_back(x);
      edge.m_yarray.push_back(y);

      // Is the current position a junction?

      // neighbor
      //  5 6 7
      //  4   0
      //  3 2 1

      int nextNeighbor;

      deg=0;

      if((pix[y][x+1] != 0)&&(neighbor != 4))
      {
            deg++;
            nextNeighbor = 0;
      }
      if((pix[y+1][x] != 0)&&(neighbor != 6))
      {
            deg++;
            nextNeighbor = 2;
```

```
        }
        if((pix[y][x-1] != 0)&&(neighbor != 0))
        {
                deg++;
                nextNeighbor = 4;
        }
        if((pix[y-1][x] != 0)&&(neighbor != 2))
        {
                deg++;
                nextNeighbor = 6;
        }
        if((pix[y+1][x+1] != 0)&&(pix[y][x+1]==0)&&(pix[y+1][x]==0)&&(neighbor != 5))
        {
                deg++;
                nextNeighbor = 1;
        }
        if((pix[y+1][x-1] != 0)&&(pix[y+1][x]==0)&&(pix[y][x-1]==0)&&(neighbor != 7))
        {
                deg++;
                nextNeighbor = 3;
        }
        if((pix[y-1][x-1] != 0)&&(pix[y-1][x]==0)&&(pix[y][x-1]==0)&&(neighbor != 1))
        {
                deg++;
                nextNeighbor = 5;
        }
        if((pix[y-1][x+1] != 0)&&(pix[y-1][x]==0)&&(pix[y][x+1]==0)&&(neighbor != 3))
        {
                deg++;
                nextNeighbor = 7;
        }

        fprintf(out, "degree=%d nextneighbor=%d\n", deg, nextNeighbor);
        neighbor = nextNeighbor;
} while(deg == 1);

// Compute the angle the edge forms at each junction (with respect to x-axis).
// 1. angle at junction 1
int chainlen = edge.m_xarray.size();

int dx = edge.m_xarray[min(4, chainlen-1)] - edge.m_xarray[0];
int dy = edge.m_yarray[min(4, chainlen-1)] - edge.m_yarray[0];

if(dx == 0)
{
        if(dy < 0)
                edge.m_angle1 = HPI;
        else
                edge.m_angle1 = -HPI;
}
else
        edge.m_angle1 = atan2((double)-dy, (double)dx);

dx = edge.m_xarray[max(chainlen-5, 0)] - edge.m_xarray[chainlen-1];
dy = edge.m_yarray[max(chainlen-5, 0)] - edge.m_yarray[chainlen-1];

if(dx == 0)
```

```
	{
		if(dy < 0)
			edge.m_angle2 = HPI;
		else
			edge.m_angle2 = -HPI;
	}
	else
		edge.m_angle2 = atan2((double)-dy, (double)dx);

	// Return the ID of the junction
	for(int i=0; i<junclist.size(); i++)
	{
		if((junclist[i]->m_x == x)&&(junclist[i]->m_y == y))
		{
			edge.m_junc2 = i;
			fclose(out);
			return i;
		}
	}

	for(i=0; i<junclist.size(); i++)
	{
		fprintf(out, "junc %d=%d %d\n", i, junclist[i]->m_x, junclist[i]->m_y);
	}
	fclose(out);
	ASSERT(FALSE);
	return -1;  // Error!!!
}


ConnectivityGraph<float>*  BinaryObjects::CreateEdgeGraph(ConnectivityGraph<Edge  *>*  jg,
ObjectInfo* objinfo)

	// Edge Graph
	ConnectivityGraph<float>*   eg   =   new   ConnectivityGraph<float>(false,   jg-
>GetNumEdges());

	//CString msg;
	//msg.Format("junction graph: number of edges=%d", jg->GetNumEdges());
	//AfxMessageBox(msg);

	// For each edge-to-edge connection, compute the angle between the edges.

	int JGsize = jg->GetSize();

	for(int i=0; i<JGsize; i++)
	{
		for(int j=i+1; j<JGsize; j++)
		{
			if(jg->IsEdge(i, j))
			{
				for(int k=0; k<JGsize; k++)
				{
					if(jg->IsEdge(j, k) && (i != k))
					{
						// Compute the angle between i,j,k

						//      i
```

```
                                    //      ^
                                    //     / \
                                    //    /   \
                                    //   /     \
                                    //  j       k

                                    if(!eg->IsEdge(jg->GetEdgeID(i, j), jg->GetEdgeID(j,
k)))
                                    {
                                          float angle = jg->GetEdge(i,j)->GetAngle(j) -
jg->GetEdge(j,k)->GetAngle(j);
                                          if(angle < 0.0f)
                                                angle += DPI;
                                          if(angle > PI)
                                                angle = DPI - angle;
                                          eg->InsertEdge(jg->GetEdgeID(i,     j),     jg-
>GetEdgeID(j,k), angle);
                                          //CString msg;
                                          //msg.Format("Inserting edge %d %d for %d %d
%d", jg->GetEdgeID(i,j), jg->GetEdgeID(j,k), i, j, k);
                                          //AfxMessageBox(msg);
                                    }
                              }
                              else if(jg->IsEdge(i, k) && (j != k))
                              {
                                    // Compute the angle between i,j,k

                                    //      j
                                    //      ^
                                    //     / \
                                    //    /   \
                                    //   /     \
                                    //  i       k

                                    if(!eg->IsEdge(jg->GetEdgeID(i, j), jg->GetEdgeID(i,
k)))
                                    {
                                          float angle = jg->GetEdge(i,j)->GetAngle(i) -
jg->GetEdge(i,k)->GetAngle(i);
                                          if(angle < 0.0f)
                                                angle += DPI;
                                          if(angle > PI)
                                                angle = DPI - angle;
                                          eg->InsertEdge(jg->GetEdgeID(i,     j),     jg-
>GetEdgeID(i, k), angle);
                                          CString msg;
                                          msg.Format("Inserting edge %d %d for %d %d %d",
jg->GetEdgeID(i,j), jg->GetEdgeID(i,k), i, j, k);
                                          AfxMessageBox(msg);
                                    }
                              }

                        }
                  }
            }
      }
      return eg;
```

```
}

ExImage* BinaryObjects::CreateJunctionImage(std::vector<ObjectInfo*> objinfo)
{
	Image* returnimg;
	returnimg = new_Image(PPM, RGB, 3, getNoOfRows_Image(m_labelImage),
		getNoOfCols_Image(m_labelImage), CVIP_BYTE, REAL);

	head_LL(m_objectlist);

	int** srcpix = (int**)m_labelImage->image_ptr[0]->rptr;
	unsigned char** destpixR = (unsigned char**)returnimg->image_ptr[0]->rptr;
	unsigned char** destpixG = (unsigned char**)returnimg->image_ptr[1]->rptr;
	unsigned char** destpixB = (unsigned char**)returnimg->image_ptr[2]->rptr;

	for(int i=0; i<size_LL(m_objectlist); i++)
	{
		Object* obj = (Object*)retrieve_LL(m_objectlist);
		for(int y = obj->y_min; y <= obj->y_max; y++)
		{
			for(int x = obj->x_min; x <= obj->x_max; x++)
			{
				if(srcpix[y][x] == obj->label)
				{
					destpixR[y][x] = 255;
					destpixG[y][x] = 255;
					destpixB[y][x] = 255;
				}
			}
		}
		linked_list* jlist = objinfo[i]->junctions;
		head_LL(jlist);
		for(int j=0; j<size_LL(jlist); j++)
		{
			Junction* junc = (Junction*)retrieve_LL(jlist);
			for(int k=0; k<junc->m_degree; k++)
			{
				int jx = junc->m_x, jy = junc->m_y;
				switch(junc->m_degree)
				{
				case 3:
					destpixR[jy][jx] = 255;
					destpixG[jy][jx] = 0;
					destpixB[jy][jx] = 0;
					break;
				case 4:
					destpixR[jy][jx] = 255;
					destpixG[jy][jx] = 255;
					destpixB[jy][jx] = 0;
					break;
				case 5:
					destpixR[jy][jx] = 0;
					destpixG[jy][jx] = 255;
					destpixB[jy][jx] = 0;
					break;
				case 6:
					destpixR[jy][jx] = 0;
```

```
                        destpixG[jy][jx] = 255;
                        destpixB[jy][jx] = 255;
                        break;
                  case 7:
                        destpixR[jy][jx] = 0;
                        destpixG[jy][jx] = 0;
                        destpixB[jy][jx] = 255;
                        break;
                  }
            }
            next_LL(jlist);
      }

      next_LL(m_objectlist);
}

ExImage *newimg = new ExImage;
newimg->m_image = returnimg;

return newimg;
```

```
void ObjectInfo::Dump()
{
      CString msg;
      msg.Format("xmin=%d  xmax=%d  ymin=%d  ymax=%d\nperimeter=%d  area=%d  eigenratio=%f
orientation=%f\nxcenter=%f ycenter=%f numjunc=%d",
            xmin, xmax, ymin, ymax, perimeter, area, eigenratio, orientation, xcenter,
ycenter, numjunc);
      AfxMessageBox(msg);
}

std::vector<ObjectInfo*> BinaryObjects::GetObjectInfo(bool buildChaincode)
{
      std::vector<ObjectInfo*> infolist;

      if(m_objectlist->listlength == 0)
            return infolist;

      getProp_Objects(m_objectlist, m_labelImage);

      head_LL(m_objectlist);

      for(int i=0; i<size_LL(m_objectlist); i++)
      {
            int ray_x;
            Object* obj = ((Object*)retrieve_LL(m_objectlist));
            //shootRay(m_labelImage, obj->label, &ray_x, &ray_y, obj->x_min, obj->y_min,
obj->x_max, obj->y_max);
            for(int c=obj->x_min; c<=obj->x_max; c++)
            {
                  if(((int*)m_labelImage->image_ptr[0]->rptr[obj->y_min])[c]    ==    obj-
>label)
                  {
                        ray_x = c;
                        break;
```

```
                }
            }

            /*
            for(int y=obj->y_min; y<=obj->y_max; y++)
            {
                for(int x=obj->x_min; x<=obj->x_max; x++)
                {
                    CString msg;
                    msg.Format("i=%d objlabel=%d, label=%d xmin=%d xmax=%d ymin=%d
ymax=%d xstart=%d ystart=%d", i, obj->label, ((int*)m_labelImage->image_ptr[0]-
>rptr[y])[x], obj->x_min, obj->x_max, obj->y_min, obj->y_max);
                    AfxMessageBox(msg);
                }
            }
            */

            ObjectInfo* info = new ObjectInfo;

            if(buildChaincode)
            {
                info->chain = new_ChainCode(obj->y_min, ray_x, obj->label);
                if(!build_LineChainCode(info->chain, m_labelImage, obj->x_min, obj-
>y_min, obj->x_max, obj->y_max))
                    AfxMessageBox("Error building chain code");
                //print_ChainCode(infolist[i]->chain, "chaininfo.txt");
                info->perimeter = info->chain->no_of_vectors;
            }
            info->area = obj->prop.area;
            info->label = obj->label;
            info->eigenratio = obj->prop.eig_ratio;
            info->orientation = obj->prop.orientation;
            info->xcenter = obj->prop.h_cog;
            info->ycenter = obj->prop.v_cog;
            info->xmin = obj->x_min;
            info->xmax = obj->x_max;
            info->ymin = obj->y_min;
            info->ymax = obj->y_max;
            //info->junctions = ComputeJunctions(info);
            infolist.push_back(info);
            next_LL(m_objectlist);
        }

        return infolist;
}

std::vector<ExImage*> BinaryObjects::CreateColorImages(std::vector<ObjectInfo*> infolist,
ImageIO* original)
{
        std::vector<ExImage*> imglist;

        int** lab = (int**)m_labelImage->image_ptr[0]->rptr;
        unsigned char** origR = original->GetPixelArray(0);
        unsigned char** origG = original->GetPixelArray(1);
        unsigned char** origB = original->GetPixelArray(2);

        for(int i=0; i<infolist.size(); i++)
```

```
	{
		ObjectInfo* info = infolist[i];
		// Create a color image for the object.
		Image* returnimg = new_Image(PPM, RGB, 3, info->ymax - info->ymin + 1,
			info->xmax - info->xmin + 1, CVIP_BYTE, REAL);

		unsigned char** destR = (unsigned char**)returnimg->image_ptr[0]->rptr;
		unsigned char** destG = (unsigned char**)returnimg->image_ptr[1]->rptr;
		unsigned char** destB = (unsigned char**)returnimg->image_ptr[2]->rptr;

		int count = 0;

		for(int y=info->ymin; y<=info->ymax; y++)
		{
			for(int x=info->xmin; x<=info->xmax; x++)
			{
				int ny = y - info->ymin;
				int nx = x - info->xmin;

				if(lab[y][x] == info->label)
				{
					destR[ny][nx] = origR[y][x];
					destG[ny][nx] = origG[y][x];
					destB[ny][nx] = origB[y][x];
					count++;
				}
				else
				{
					destR[ny][nx] = (unsigned char)0;
					destG[ny][nx] = (unsigned char)0;
					destB[ny][nx] = (unsigned char)0;
				}
			}
		}
		ExImage* newimage = new ExImage;
		newimage->m_image = returnimg;
		imglist.push_back(newimage);
	}
	return imglist;
}

ExImage* BinaryObjects::CreateColorImage(ObjectInfo* info, ImageIO* original)
{

	int** lab = (int**)m_labelImage->image_ptr[0]->rptr;
	unsigned char** origR = original->GetPixelArray(0);
	unsigned char** origG = original->GetPixelArray(1);
	unsigned char** origB = original->GetPixelArray(2);

		// Create a color image for the object.
		Image* returnimg = new_Image(PPM, RGB, 3, info->ymax - info->ymin + 1,
			info->xmax - info->xmin + 1, CVIP_BYTE, REAL);

		unsigned char** destR = (unsigned char**)returnimg->image_ptr[0]->rptr;
		unsigned char** destG = (unsigned char**)returnimg->image_ptr[1]->rptr;
		unsigned char** destB = (unsigned char**)returnimg->image_ptr[2]->rptr;
```

```
        int count = 0;

        for(int y=info->ymin; y<=info->ymax; y++)
        {
              for(int x=info->xmin; x<=info->xmax; x++)
              {
                    int ny = y - info->ymin;
                    int nx = x - info->xmin;

                    if(lab[y][x] == info->label)
                    {
                          destR[ny][nx] = origR[y][x];
                          destG[ny][nx] = origG[y][x];
                          destB[ny][nx] = origB[y][x];
                          count++;
                    }
                    else
                    {
                          destR[ny][nx] = (unsigned char)0;
                          destG[ny][nx] = (unsigned char)0;
                          destB[ny][nx] = (unsigned char)0;
                    }
              }
        }
        ExImage* newimage = new ExImage;
        newimage->m_image = returnimg;

        return newimage;


std::vector<Junction*> BinaryObjects::ComputeJunctions(ObjectInfo* objinfo)

    std::vector<Junction*> junclist;

    int** pix = (int**)m_labelImage->image_ptr[0]->rptr;
    int label = objinfo->label;

    for(int y=objinfo->ymin; y<=objinfo->ymax; y++)
          for(int x=objinfo->xmin; x<=objinfo->xmax; x++)
          {
                std::set<int> neighbor;

                if(pix[y][x] == label)
                {
                      // neighbor
                      //  5 6 7
                      //  4   0
                      //  3 2 1

                      int deg=0;
                      if((x!=objinfo->xmax) && (pix[y][x+1] == label))
                      {
                            deg++;
                            neighbor.insert(0);
                      }
                      if((y!=objinfo->ymin) && (pix[y-1][x] == label))
                      {
```

```
                        deg++;
                        neighbor.insert(6);
                    }
                    if((x!=objinfo->xmin) && (pix[y][x-1] == label))
                    {
                        deg++;
                        neighbor.insert(4);
                    }
                    if((y!=objinfo->ymax) && (pix[y+1][x] == label))
                    {
                        deg++;
                        neighbor.insert(2);
                    }
                    if((y!=objinfo->ymin) && (x!=objinfo->xmax) && (pix[y-1][x+1] ==
label))
                    {
                        if((neighbor.find(6)==neighbor.end())                    &&
(neighbor.find(0)==neighbor.end()))
                        {
                            deg++;
                            neighbor.insert(7);
                        }
                    }
                    if((y!=objinfo->ymin) && (x!=objinfo->xmin) && (pix[y-1][x-1] ==
label))
                    {
                        if((neighbor.find(4)==neighbor.end())                    &&
(neighbor.find(6)==neighbor.end()))
                        {
                            deg++;
                            neighbor.insert(5);
                        }
                    }
                    if((y!=objinfo->ymax) && (x!=objinfo->xmin) && (pix[y+1][x-1] ==
label))
                    {
                        if((neighbor.find(2)==neighbor.end())                    &&
(neighbor.find(4)==neighbor.end()))
                        {
                            deg++;
                            neighbor.insert(3);
                        }
                    }
                    if((y!=objinfo->ymax) && (x!=objinfo->xmax) && (pix[y+1][x+1] ==
label))
                    {
                        if((neighbor.find(0)==neighbor.end())                    &&
(neighbor.find(2)==neighbor.end()))
                        {
                            deg++;
                            neighbor.insert(1);
                        }
                    }

                    // Modify neighborhood list based on the following rules:
                    // if the neighbor is odd and there is an adjacent neighbor,
                    // remove the neighbor from the list.
```

```
                        /*
                        std::set<int>::iterator it = neighbor.begin();
                        int c = 0;
                        while(it != neighbor.end())
                        {
                              if(*it % 2 == 1)
                              {
                                    int n = *it;
                                    int n1 = (n != 0) ? n-1 : 7;
                                    int n2 = (n != 7) ? n+1 : 0;
                                    if((neighbor.find(n1)     !=     neighbor.end())     ||
(neighbor.find(n2) != neighbor.end()))
                                    {
                                          it = neighbor.erase(it);
                                          deg--;
                                    }
                              }
                              else
                                    it++;
                              if(c>10)
                                    AfxMessageBox("ERROR");
                              c++;
                        }
                        */
                        if(deg != 2)
                        {
                              Junction *junc = new Junction;
                              junc->m_degree = deg;
                              junc->m_x = x;
                              junc->m_y = y;
                              junc->m_neighborPixels = neighbor;
                              junclist.push_back(junc);
                        }
                  }
            }
      return junclist;
}


void     BinaryObjects::MergeJunctions(ConnectivityGraph<Edge     *>     *jgraph,     float
mergeDistance)
{
      // juncs keeps track of junctions that belong to "short" edges.
      std::set<int> juncs;
      std::set<int>::iterator it;

      Point* juncpos = new Point[jgraph->GetSize()];

      for(int i=0; i<jgraph->GetSize(); i++)
      {
            // Find edges that are short enough and put their junctions into juncs.
            // Remove those edges.
            for(int j=i+1; j<jgraph->GetSize(); j++)
            {
                  if(jgraph->IsEdge(i, j))
                  {
```

```
                        if(jgraph->GetEdge(i, j)->m_length < mergeDistance)
                        {
                              juncpos[i] = jgraph->GetEdge(i, j)->GetEndPosition(i);
                              juncs.insert(i);
                              juncs.insert(j);
                              jgraph->DeleteEdge(i, j);
                        }
                  }
            }
      }

std::vector<std::vector<int> > groups;

// Look at junctions' connectivity and divide them into groups
// of connected junctions.
while(!juncs.empty())
{
      std::vector<int> list;
      groups.push_back(list);

      int current = *juncs.begin();
      groups.rbegin()->push_back(current);
      juncs.erase(juncs.begin());

      std::vector<int> worklist;
      do
      {
            it = juncs.begin();
            while(it!=juncs.end())
            {
                  if(jgraph->IsEdge(current, *it))
                  {
                        worklist.push_back(*it);
                        it = juncs.erase(it);
                  }
                  else
                        it++;
            }
            if(worklist.empty())
                  break;
            current = *worklist.begin();
            worklist.erase(worklist.begin());
            groups.rbegin()->push_back(current);
      } while(!worklist.empty());
}

int* newjuncIndex = new int[groups.size()];
Point* newjuncPos = new Point[groups.size()];

// Compute the centroid for each group and make it a new junction.
for(i=0; i<groups.size(); i++)
{
      float sumx = 0.0f;
      float sumy = 0.0f;
      for(int j=0; j<groups[i].size(); j++)
      {
            sumx += juncpos[groups[i][j]].x;
```

```
                sumy += juncpos[groups[i][j]].y;
        }
        sumx /= groups[i].size();
        sumy /= groups[i].size();
        Point pt(ROUND(sumx), ROUND(sumy));
        newjuncPos[i] = pt;
        newjuncIndex[i] = groups[i][0];
    }

    for(i=0; i<groups.size(); i++)
    {
        for(int j=0; j<groups[i].size(); j++)
        {
            // Modify all edges that are connected to groups[i][j]
            for(int k=0; k<jgraph->GetSize(); k++)
            {
                if(jgraph->IsEdge(groups[i][j], k))
                {
                    Edge* edge = jgraph->GetEdge(groups[i][j], k);
                    //AddSegmentEnd(edge, newjuncPos[i], newjuncIndex[i], k,
mergeDistance);

                    // Compute the angle the edge forms at each junction (with
respect to x-axis).
                    // 1. angle at junction 1
                    int chainlen = edge->m_xarray.size();

                    int endIndex;

                    if(newjuncIndex[i] == edge->m_junc1)
                        endIndex = 0;
                    else
                        endIndex = chainlen - 1;

                    int dx = newjuncPos[i].x - edge->m_xarray[endIndex];
                    int dy = newjuncPos[i].y - edge->m_yarray[endIndex];

                    if(dx == 0)
                    {
                        if(dy > 0)
                        {
                            if(newjuncIndex[i] == edge->m_junc1)
                                edge->m_angle1 = HPI;
                            else
                                edge->m_angle2 = HPI;
                        }
                        else
                        {
                            if(newjuncIndex[i] == edge->m_junc1)
                                edge->m_angle1 = -HPI;
                            else
                                edge->m_angle2 = -HPI;
                        }
                    }
                    else
                    {
                        if(newjuncIndex[i] == edge->m_junc1)
```

```
                                        edge->m_angle1        =        atan2((double)-dy,
(double)dx);
                                  else
                                        edge->m_angle2        =        atan2((double)-dy,
(double)dx);
                              }
                        }
                  }
            }
      }
}

void BinaryObjects::AddSegmentEnd(Edge* edge, Point newPos, int newIndex, int otherIndex,
float mergeDistance)
{
      // Connect the edge to the newly formed junction.

      ASSERT(edge->m_junc1 == otherIndex || edge->m_junc2 == otherIndex);
      if(edge->m_junc1 == otherIndex)
            edge->m_junc2 = newIndex;
      else
            edge->m_junc1 = newIndex;




/**
// Helper function for GetObjectInfo.
void  BinaryObjects::ComputeJunctions(ObjectInfo*  objinfo,  Junction  **junctions,  int
&numjunc)
{
      int xmin = objinfo->xmin;
      int ymin = objinfo->ymin;
      int xmax = objinfo->xmax;
      int ymax = objinfo->ymax;

      // Convert the chain code into a list of (x,y).
      int *xP=NULL, *yP=NULL;
      getXY_ChainCode(objinfo->chain, &xP, &yP);
      // count is a counter for each coordinate in the chain code.
      int** count = new int*[ymax-ymin+1];
      for(int i=0; i<ymax-ymin+1; i++)
      {
            count[i] = new int[xmax-xmin+1];
            // Reset counter.
            for(int j=0; j<xmax-xmin+1; j++)
                  count[i][j] = 0;
      }

      numjunc = 0;

      for(i=0; i<objinfo->chain->no_of_vectors; i++)
      {
            // Increment numjunc if the junction degree is greater than 2.
            if(count[yP[i]-ymin][xP[i]-xmin] == 2)
            {
                  numjunc++;
```

```
        }
        (count[yP[i]-ymin][xP[i]-xmin])++;
    }

    // No junction.
    if(numjunc == 0)
    {
        *junctions = NULL;
        return;
    }

    Junction* newjunctions = new Junction[numjunc];

    int jc = 0;
    for(i=0; i<ymax-ymin+1; i++)
        for(int j=0; j<xmax-xmin+1; j++)
        {
            if(count[i][j] >= 3)
            {
                newjunctions[jc].degree = count[i][j];
                newjunctions[jc].x = j+xmin;
                newjunctions[jc].y = i+ymin;
                jc++;
            }
        }
    *junctions = newjunctions;
    ASSERT(jc == numjunc);

    // Clean up.
    delete [] xP;
    delete [] yP;
*/

typedef HashTable *ObjectHash;

#define HASH_SIZE 251U

#define hash_Object(label) (((unsigned)(label))%HASH_SIZE)

static void addto_Objects(ObjectHash hashP, int next_label, Color pixel, int y_pos, int
x_pos);
static void update_Objects(ObjectHash hashP, int object_label, int y_pos, int x_pos);
static void combine_Objects(ObjectHash hashP, int b, int c, int *xmin, int *xmax, int
*ymin, int *ymax);
static ObjectList hash2List_Objects(ObjectHash hashP);

static void makegraymap_Objects(ColorHistogram *chP);
static Matrix *color2index_Image(ColorHistogram *chP, Image *imageP );

/*
 * label recycling routines
 */
static void initLabelStack(void);
static int getNextLabel(void);
static void recycleLabel(int label);
```

```
/*
 * global variables used by label recycling routines
 */
static int label_count;
static Stack label_stackP;

ObjectList
label_Objects2(
                    Image *imageP,
                    Image **labelP,
                    unsigned background
                    )
{
      register int x, y=0, i, j;
      int A, B, C, D, rows, cols;
      byte *pixarray2, *pixarray1; /* pixel arrays */
      int *labarray1, *labarray2, *rowP; /* label arrays */
      int xmin, xmax, ymin, ymax, next_label;
      unsigned *A_LABEL, B_LABEL, C_LABEL, D_LABEL, MAX_LABEL;
      ObjectList listP;
      ObjectHash hashP;
      Object *objP;
      ColorHistogram *chP;
      ColorHistObject *mapP;
      Matrix *matrixP;
      ROI *roiP;
      const char *fn = "label";

      chP = new_ColorHist();

      if(getNoOfBands_Image(imageP) > 1) {
            compute_ColorHist(chP, imageP, 256);
            matrixP = color2index_Image(chP, imageP);

      }
      else {
            makegraymap_Objects(chP);
            matrixP = getBand_Image(imageP,0);
      }

      mapP = chP->histogram;
      if(mapP==NULL) return NULL;

      rows = getNoOfRows_Image(imageP);
      cols = getNoOfCols_Image(imageP);

      *labelP = new_Image(PGM, GRAY_SCALE, 1, rows, cols, CVIP_INTEGER, REAL);

      pixarray2 = (unsigned char*)getRow_Matrix(matrixP, 0);
      labarray2 = (int *)getRow_Image(*labelP, 0, 0);

      initLabelStack();
      hashP = new_HT(HASH_SIZE);

      /*
      * handle special case of first row
      */
```

```
for(x=0; x < cols; x++)
      if( pixarray2[x] != background ) {
            if( (x==0) || (pixarray2[x-1] != pixarray2[x]) ) {
                  next_label = getNextLabel();
                  addto_Objects(hashP, next_label, mapP[pixarray2[x]].pixel, y, x);
                  labarray2[x] = next_label;
            }
            else {
                  update_Objects(hashP, labarray2[x-1], y, x);
                  labarray2[x] = labarray2[x-1];
            }
      }

      for(y=0; y < rows-1; y++)
      {

            pixarray1 = pixarray2;
            pixarray2 = (unsigned char*)getRow_Matrix(matrixP, y+1);

            labarray1 = labarray2;
            labarray2 = (int *)getRow_Image(*labelP, y+1, 0);

            for(x=0; x < cols-1; x++)
            {
                  A = pixarray2[x+1];
                  B = pixarray2[x];
                  C = pixarray1[x+1];
                  D = pixarray1[x];

                  A_LABEL = (unsigned *) &labarray2[x+1];
                  B_LABEL = labarray2[x];
                  C_LABEL = labarray1[x+1];
                  D_LABEL = labarray1[x];

                  if (x == 0)
                  {
                        if (B != background)
                              if(D == B)
                              {
                                    B_LABEL = labarray2[x] = D_LABEL;
                                    update_Objects(hashP, B_LABEL, y+1, x);
                              }
                              else
                              {
                                    next_label = getNextLabel();
                                    addto_Objects(hashP,                    next_label,
mapP[B].pixel,y+1,x);
                                    B_LABEL = labarray2[x] = next_label;
                              }
                  } // x==0
                  if (A != background)
                  {
                        if(D != A)
                        {
                              if(B != A)
                              {
```

```
                                    if(C != A)
                                    {
                                          next_label = getNextLabel();
                                          addto_Objects(hashP,
next_label,mapP[A].pixel, y+1, x+1);
                                          *A_LABEL = next_label;
                                    } // C!=A
                                    else
                                    {
                                          *A_LABEL = C_LABEL;
                                          update_Objects(hashP,    *A_LABEL,    y+1,
x+1);
                                    } // C==A
                              } // B!=A
                              else
                              {
                                    if (C == A)
                                    {
                                          if (B_LABEL == C_LABEL)
                                          {
                                                *A_LABEL = B_LABEL;
                                          }
                                          else
                                          {
                                                combine_Objects(hashP,      B_LABEL,
C_LABEL, &xmin, &xmax,
                                                      &ymin, &ymax);
                                                MAX_LABEL = MAX(B_LABEL,C_LABEL);
                                                *A_LABEL =  MIN(B_LABEL,C_LABEL);
                                                roiP = new_ROI();
                                                asgnImage_ROI(roiP,  *labelP,  xmin,
ymin, xmax-xmin+1,
                                                      ymax-ymin+1);
                                                for(i=0; i < getNoOfRows_ROI(roiP);
i++) {
                                                      rowP      =      (int      *)
getRow_ROI(roiP, i, 0);
                                                      for(j=0;          j          <
getNoOfCols_ROI(roiP); j++, rowP++)
                                                            if(*rowP == MAX_LABEL)
                                                                  *rowP = *A_LABEL;
                                                }
                                                delete_ROI(roiP);
                                          } // B_LABEL!=C_LABEL
                                    } // C==A
                                    else
                                          *A_LABEL = B_LABEL;

                                    update_Objects(hashP, *A_LABEL, y+1, x+1);
                              } // B!=A
                        }  // D!=A
                        else
                        {
```

```
                        *A_LABEL = D_LABEL;
                        update_Objects(hashP, *A_LABEL, y+1, x+1);
                    } // D==A
                } // A!=background
                else  // addition by Yusaku Sako. 7/26/99
                {
                    if(B != background)
                        if((B == C) && (D != B) && (B_LABEL != C_LABEL))
                        {
                            combine_Objects(hashP, B_LABEL, C_LABEL, &xmin,
&xmax,
                                &ymin, &ymax);

                            MAX_LABEL = MAX(B_LABEL,C_LABEL);
                            B_LABEL = labarray2[x] = MIN(B_LABEL,C_LABEL);

                            roiP = new_ROI();
                            asgnImage_ROI(roiP, *labelP, xmin, ymin, xmax-
xmin+1,
                                ymax-ymin+1);

                            for(i=0; i < getNoOfRows_ROI(roiP); i++) {
                                rowP = (int *) getRow_ROI(roiP, i, 0);
                                for(j=0; j < getNoOfCols_ROI(roiP); j++,
rowP++)
                                    if(*rowP == MAX_LABEL)
                                        *rowP = B_LABEL;
                            }

                            delete_ROI(roiP);
                            update_Objects(hashP, B_LABEL, y+1, x); //
erase
                        }
                } // A==background
            } // for x
        } // for y

        if(hashP->table[hashP->key])
            listP = hash2List_Objects(hashP);
        else
            listP = NULL;

        /*
        fprintf(stderr, "Displaying Object Statistics...\n\n");
        fprintf(stderr, "Number of objects found = %u.\n\n", size_LL(listP));
        print_Objects(listP);
        fprintf(stderr, "\n\n                    Press the <ENTER> key to continue");
        getchar();
        */

        // Bug fix by YS (3/24/00).
        // Don't wanna delete matrixP when number of bands is 1
        if(getNoOfBands_Image(imageP) > 1) {
            delete_Matrix(matrixP);
        }

        return listP;
```

```
}

static ObjectList hash2List_Objects(ObjectHash hashP)
{
   register int i, first = 0;
   ObjectList listP;

   for(;;) {
      if(hashP->table[first]) break;
      first++;
   }

   listP = hashP->table[first];

   for(i=first+1; i < size_HT(hashP); i++)
      if(hashP->table[i]) {

         *(listP->tailP) = *(hashP->table[i]->headP->nextP);
         listP->tailP = hashP->table[i]->tailP;
         listP->listlength += size_LL(hashP->table[i]);
       delete_Link(hashP->table[i]->headP->nextP);
         delete_Link(hashP->table[i]->headP);
         /* delete_Link(hashP->table[i]->headP->nextP); */
         free(hashP->table[i]);

       }

    return listP;
}

static void addto_Objects(ObjectHash hashP, int next_label, Color pixel, int y_pos, int
x_pos)

   setKey_HT(hashP, hash_Object(next_label));
   addObject_HT( hashP, new_Object(next_label, pixel, y_pos, x_pos) );


static void update_Objects( ObjectHash hashP, int object_label, int y_pos, int x_pos)
{
   Object *obj;
   const char *fn = "update_Objects";

   setKey_HT(hashP,hash_Object(object_label));

   if(findObject_HT(hashP, match_Object, &object_label)) {
      obj = (Object*)getObject_HT(hashP);

      obj -> x_min = MIN(obj -> x_min, x_pos);
      obj -> x_max = MAX(obj -> x_max, x_pos);
      obj -> y_min = MIN(obj -> y_min, y_pos);
      obj -> y_max = MAX(obj -> y_max, y_pos);
   }
}


static void combine_Objects(ObjectHash hashP, int b, int c, int *xmin, int *xmax, int
*ymin, int *ymax)
```

```
{
   dlink *linkP;
   Object *bP, *cP;

   if (b < c)
   {
         int temp;
         temp = b;
         b = c;
         c = temp;
   }

   setKey_HT(hashP,hash_Object(b));
   findNextObject_HT(hashP, match_Object, &b);
   bP = (Object*)getNextObject_HT(hashP);
   removeNextObject_HT(hashP);

   setKey_HT(hashP,hash_Object(c));
   findObject_HT(hashP, match_Object, &c);
   cP = (Object*)getObject_HT(hashP);

   *xmin = bP->x_min;
   *xmax = bP->x_max;
   *ymin = bP->y_min;
   *ymax = bP->y_max;

   cP -> x_min = MIN(cP -> x_min, bP->x_min);
   cP -> x_max = MAX(cP -> x_max, bP->x_max);
   cP -> y_min = MIN(cP -> y_min, bP->y_min);
   cP -> y_max = MAX(cP -> y_max, bP->y_max);

   delete_Object(bP);

   recycleLabel(b);


static void makegraymap_Objects( ColorHistogram *chP )
{
   register int i;

   chP->histogram = (ColorHistObject *) malloc(256*sizeof(ColorHistObject));

   for(i=0; i < 256; i++)
      assign_Color(chP->histogram[i].pixel, i, i, i);
}


static Matrix *color2index_Image(ColorHistogram *chP, Image *imageP )
{
   Matrix *matrixP;
   Color pixel;
   ColorHashTable *chtP;
   unsigned rows, cols;
   register int i;
   byte *mP, *rP, *gP, *bP;
```

```
    if(chP->no_of_colors > 256) {
       return NULL;
    }

    rows = getNoOfRows_Image(imageP);
    cols = getNoOfCols_Image(imageP);
    rP = (unsigned char*)getRow_Image(imageP, 0, RED);
    gP = (unsigned char*)getRow_Image(imageP, 0, GRN);
    bP = (unsigned char*)getRow_Image(imageP, 0, BLU);

    matrixP = new_Matrix(rows, cols, CVIP_BYTE, REAL);
    mP = (unsigned char*)getRow_Matrix(matrixP,0);

    chtP = hist2Hash_ColorHT(chP);

    for(i=0; i < rows*cols; i++, mP++, rP++, gP++, bP++) {

       assign_Color(pixel, *rP, *gP, *bP);

       if( (*mP = lookUpColor_ColorHT( chtP, pixel )) == -1 )
          return NULL;
    }
    delete_ColorHT(chtP);

    return matrixP;



static void initLabelStack(void)
{
       label_stackP = new_Stack();
       label_count=1;


static int getNextLabel(void)
{
       int next_label, *labelP;

       if(isempty_Stack(label_stackP))
             next_label = label_count++;
       else {
             labelP = (int*)pop_Stack(label_stackP);
             next_label = *labelP;
             free(labelP);
       }

       return next_label;
}

static void recycleLabel(int label)
{
       int *labelP = (int *) malloc(sizeof(int));
       *labelP = label;
       push_Stack(label_stackP, labelP);
}
```

```
// Connectedness.  Used in ImageThinning function.
int cconc(int inb[9])
{
      int icn = 0;

      for(int i=0; i<8; i+=2)
      {
            if(inb[i]==0)
                  if(inb[i+1] == 255 || inb[i+2] == 255)
                        icn++;
      }
      return icn;
}

// Performs thinning of a binary image
Image* ImageThinning(Image *img)
{
      // Copy original image
      Image *newimg = duplicate_Image(img);
      if(!newimg)
            AfxMessageBox("Not enough memory!");
      unsigned char **pix = (unsigned char **)newimg->image_ptr[0]->rptr;
      int m = 100;  int ir = 1, ia[9], ic[9];

            int numrows = img->image_ptr[0]->rows;
            int numcols = img->image_ptr[0]->cols;


      while(ir!=0)
      {
            ir=0;

            for(int iy=0; iy < numrows; iy++)
            {
                  for(int ix=0; ix < numcols; ix++)
                  {
                        if(pix[iy][ix] != 255)
                              continue;

                        if(ix == numcols-1)
                              ia[0] = 0;
                        else
                              ia[0] = pix[iy  ][ix+1];
                        if(iy == 0 || ix == numcols-1)
                              ia[1] = 0;
                        else
                              ia[1] = pix[iy-1][ix+1];
                        if(iy == 0)
                              ia[2] = 0;
                        else
                              ia[2] = pix[iy-1][ix  ];
                        if(iy == 0 || ix == 0)
                              ia[3] = 0;
                        else
                              ia[3] = pix[iy-1][ix-1];
                        if(ix == 0)
                              ia[4] = 0;
```

```
else
      ia[4] = pix[iy  ][ix-1];
if(iy == numrows-1 || ix == 0)
      ia[5] = 0;
else
      ia[5] = pix[iy+1][ix-1];
if(iy == numrows-1)
      ia[6] = 0;
else
      ia[6] = pix[iy+1][ix  ];
if(iy == numrows-1 || ix == numcols-1)
      ia[7] = 0;
else
      ia[7] = pix[iy+1][ix+1];

for(int i=0; i < 8; i++)
{
      if(ia[i] == m)
      {
            ia[i] = 255; ic[i] = 0;
      }
      else
      {
            if(ia[i] < 255)
                  ia[i] = 0;
            ic[i] = ia[i];
      }
}  // for
ia[8] = ia[0];
ic[8] = ic[0];
if(ia[0]+ia[2]+ia[4]+ia[6]==255*4)
      continue;

int iv, iw;
for(i=0, iv=0, iw=0; i<8; i++)
{
      if(ia[i]==255)
            iv++;
      if(ic[i]==255)
            iw++;
}
if(iv<=1)
      continue;
if(iw==0)
      continue;
if(cconc(ia)!=1)
      continue;

int temppix = (iy == 0) ? 0 : pix[iy-1][ix];
if(temppix == m)
{
      ia[2] = 0;
      if(cconc(ia) != 1)
            continue;
      ia[2] = 255;
}
temppix = (ix == 0) ? 0 : pix[iy][ix-1];
```

```
                        if(temppix == m)
                        {
                              ia[4]=0;
                              if(cconc(ia)!=1)
                                    continue;
                              ia[4] = 255;
                        }
                        pix[iy][ix] = m;
                        ir++;
                  }  // for ix
            }  // for iy
            m++;
      }  // while
      for(int iy=0; iy<img->image_ptr[0]->rows; iy++)
      {
            for(int ix=0; ix<img->image_ptr[0]->cols; ix++)
            {
                  if(pix[iy][ix] < 255)
                        pix[iy][ix] = 0;
            }
      }
      return newimg;
```